X2000

Software Architecture Definition

June 15, 1997

A Cooperative Effort
by the Unified Flight / Ground Architecture Team

Contents

o

1 Introduction

Unmanned exploration places peculiar demands on the systems we send to other planets and the remote reaches of space. They must be extremely reliable in the face of an often poorly understood environment. They must operate for long stretches without intervention. And they must be self-sufficient in their dealings with failures and other calamities. The extent of our exploration is limited by the level of autonomy that we can endow. This in turn depends on the sophistication and adaptability of the software that serves as our proxy, managing these systems and carrying out the desired tasks.

In the future we would like to move into closer contact with the objects of space exploration, exploring surfaces, atmospheres, and other realms, fielding more complex instruments, and moving farther into distant space. More ambitious missions push greater demands upon this software.

Managing the resulting complexity has forced us to abandon our well worn but limiting approach to software design for these systems. It has also become imperative to reduce the cost and time spent on their development. Both considerations have resulted in the creation of a task to generate a new software architecture for future missions. This task has been initiated by the X2000 program and is the subject of this document.

o

2 Document Objectives

This document was conceived to describe the top level goals and technical approach for a Unified Flight/Ground Architecture (UFGA) for software [See “Software Architecture” in Appendix A — Definitions]. It is not a specification from which a particular design might be produced, but rather is meant to expose issues that potential designs must address, to promote general principles and features deemed important by its contributors, and to suggest a general framework that may accomplish these objectives.

The intended scope of this effort is broad, including both engineering and instrument flight software, ground software for spacecraft operation and monitoring, and test software for all levels of integration. The unifying theme among these areas is that they all play an operative role in the activities of a spacecraft or other remote vehicle at some time during its life span.

This document is divided into sections which address major aspects of design, beginning with general design guidelines. This is followed by a discussion of general software

methods that apply at all levels. Then the issues associated with each of several particular functional areas is addressed.

This is a draft version of the document. Comments, criticisms (graciously offered), and contributions to its content are welcome.

o

3 Design Guidelines

The following objectives do not in themselves directly dictate any software architecture. They state high level attributes which would be achieved ideally (in the view of the contributors) by any design meeting the programmatic and technical challenges facing future space exploration.

3.1 Support a Wide and Increasingly Challenging Range of Missions

Many missions in the near future will still involve flying by bodies, or rendezvousing with them and orbiting around them. Eventually, landers and surface rovers will become more common, and other sorts of mobile platforms, such as balloons or even submarines, may be necessary. Sample return vehicles may be required to bring the rewards home. These systems may work in partnership with orbiters or others of their kind. In interplanetary space, large aperture experiments involving closely coordinated groups of spacecraft will be flying.

These missions vary in the degree of ground visibility and control that is possible, the complexity and immediacy of the required tasks, the uncertainty of the environment with which they must contend, the reliability they must achieve, and the limitations of their physical resources. They will also vary in the level of technology available at their inauguration, the extent to which inherited elements impose constraints on the design, and the cost and schedule available for implementation. These features all bear directly and substantially on the software capabilities required to support them. A painful consequence has been the wasteful proliferation of systems and approaches populating past programs.

One measure of success of the UFGA will be the range of missions it is able to support, especially if its utilization arises, not as an imposition, but rather as a result of free selection by implementers for its attractive features. The architecture must therefore have attributes conducive to wide applicability. These include the following.

3.1.1 Flexible and Adaptable Software Components

The architecture must permit a variety of implementations so it may be tailored cleanly to particular applications.

At the software component level this means that the following adjustments should be relatively easy:

- The capabilities of each component can be adjusted to meet related but different needs.
- Alternate implementations can be substituted within each area of functionality.
- Nonessential functions can be removed.
- New components, not previously anticipated, can be added.

One essential factor is choosing the granularity of the functional division such that separable functions are not intermingled in the same component. Another is avoiding duplication among components such that consistency is difficult to maintain, while avoiding a design that unnecessarily couples component implementations.

3.1.2 Adjustable Hardware and Software Infrastructure

In addition, elements of the architecture infrastructure must also be adjustable. It should move easily across a variety of computing platforms where any of the following might change (within certain prescribed but not overly tight constraints):

- Processor speed, type, and quantity
- Memory and mass storage architecture
- Networking, signaling, and general I/O architecture
- Core operating system
- Operating system support structures

3.1.3 Capability in Layered Increments

The UFGA must be able to support elaborate missions in uncertain environments at one extreme while also capturing relatively simple applications with the least of requirements. This is a broad spectrum for which no single paradigm applies. As complexity is added, the necessary structure required to organize and manage problems can shift dramatically in character. We see this in real world examples, such as biological and economic systems, where nested layers of control have evolved, each manifesting itself in different ways, from chemical processes at one end to governmental regulation at the other. Nevertheless, all of these layers interact in a structured manner that productively unifies the entire collection.

Rather than identifying a suite of architectures with each most suitable to a restricted range of application, we prefer to learn from natural examples by proposing a unified architecture that spans the entire range, to be accomplished by partitioning functionality in a layered manner and applying the appropriate paradigm to each. *["This statement, at this point in the document, needs more justification to be compelling. A hierarchical or layered organization is not necessarily better than an organization of interacting agents, but it may be a more comfortable model. Actually, as I read in later sections, I think you have a very good rationale for layered organization, and maybe it just needs to be summarized here." — Dvorak]* This must reflect communication and control issues for a broad range of remote mission complexity and it must facilitate functional migration. Consequently, the architecture should be completely viable with just the lowest layer (or two, or three, …) residing on the fielded system (and its subordinates, if any).

In this manner the architecture captures even the simplest remote systems, which may then be viewed merely as harboring the lowest level components of a full UFGA architecture, the bulk of which happens to reside elsewhere.

At the opposite extreme, a fully autonomous remote system which receives only very high level instruction from the ground can also have essentially the same full UFGA architecture, except that in this case the bulk of it resides on board. Nevertheless, the lowest components in this interconnected system should share the same basic role as their counterparts in simple systems.

Intermediate cases can also be envisioned. For instance, partial autonomy results where all but the highest levels of the system fly. In another example, one or more low level remote systems (e.g., some aerobots) may be guided from a central remote site (e.g., an orbiter) with more autonomous capability. The strength of the architecture will be in its ability to divide and realign as necessary to meet these various needs.

3.2 Unifying Paradigms

A few useful principles can guide an effective layered architecture.

3.2.1 Goal Directed Behavior

In a layered architecture each layer in performing its functions depends in large measure on discounting the fine details of lower layers, including the uncertainty and incomplete knowledge confronting them. To make this possible, each component in the lower layers must perform actions which make its behavior predictable in some bounding sense, despite the difficulties. It must continually correct its behavior in response to perceived events, presenting a somewhat idealized behavior to layers above which are not privy to the numerous actions required to mask the uncertainties. The resulting actions are therefore not directly dictated by the higher layer, but rather fall as a consequence of a more abstract goal established by the higher layer in conjunction with the conditions encountered.

This reflexive application of local feedback is necessary at each layer until at the highest layer operators of the system become the goal providers. Any compromise to this goal directed commanding at each layer by directly intervening in lower layers with explicitly commanded actions can only create havoc with the overall architecture.

3.2.2 Weak Coupling

Peer level interactions with other components are among the uncertainties with which each component must cope. As peers there is less opportunity to react fruitfully to the behavior of other components for which no direct control may be possible. Attempts to do so may become chaotic, or unstable, and deadlocks can arise where little progress is made. The role of higher layers is to put some form on these interactions, but this becomes unmanageable if the interactions are numerous or complex. Therefore, coupling among components must be minimized by making components tolerant to defects in their peers and by requiring components to contain the effects of local problems if possible.

3.3 Facilitate Integration and Test

Systems to be captured by the UFGA are complex. They consist of numerous components spanning flight, operations, and test systems. It is important that these systems not be designed in a manner that requires large portions of the system to be present for meaningful integration and test to occur. It is also important that the verification of the system be largely possible through the independent assessment of subsets at each level of integration without having to repeat these verifications for the system at large. Otherwise, enormous analytical effort is required throughout the development to assure the success of a late, intense, and usually unsatisfying integration.

3.3.1 Parallel Development and Test of Components

Facilitating integration and test requires a reasonably decoupled design with software components that are self-contained to the extent possible. Once this has been accomplished, however, the actions of a component must be carefully delineated such that there is never any ambiguity in the state of the component, and such that each action on the component moves it unambiguously from one state to another. To achieve this, a discipline must be imposed on the implementation of each component, ideally through the tools used to create it. This will enable a rigorous definition of the actions of each component that can then be tested and verified at the component level.

3.3.2 Layered Operation

Once avoidable dependencies have been eliminated, it may still not be easy to test a component in isolation, especially when a complex sequence of interactions are involved that are difficult to generate analytically. To create a realistic test, each component generally requires some representation of the interfaces above that control it, and often requires a fairly accurate dynamical representation of its peers and of all the components below it that it controls. Various strategies for dealing with this problem can be enhanced if the architecture supports them.

Interface Visibility

It should be possible to intercept data at all interfaces for examination. This should include enough information to expose relationships between traffic on different interfaces and activities in any encompassing environment.

Incremental Assembly

It should be possible to assemble the architecture from the lowest layers up, missing upper layers being replaced by test drivers enabling explicit exercise of the layers present. This has a number of benefits:

- It aids testing by limiting the scope in the beginning to a manageable level, and allowing each subsequent layer to build on a solid, verified foundation.
- It enhances the level of control available for testing components in each layer rather, than having to rely on indirect means.
- It gets integration started earlier when problems are easier to correct.

- It provides more programmatic flexibility for deliveries.
- It encourages the layering approach desired for reasons of adaptability, as described above.

Variable Fidelity Execution

It should also be possible to assemble the architecture from the highest layers down, missing lower layers being replaced by reduced fidelity models of behavior. This has a number of benefits as well:

- It permits a far larger number of scenarios to be run on high level functions for which the details of operation at low levels are generally not (and should not be) important.
- It provides a basis for comparison against which the behavior of lower level functions can be verified.
- It enables the architecture to play a feasible role in higher level simulations of mission activities.
- It complements and directly benefits from the model based design methods described below.

Substitution Testing

It should be possible to assemble the architecture on the "flight" hardware, on a simulation of the "flight" hardware, or on a relatively arbitrary mixture of the two. For missing hardware elements, the level of replacement should be selectable, taking place directly at the hardware interface or higher into the software hierarchy, as appropriate. The benefits of this are as follows:

- Hardware interfaces can be tried early, before "flight" hardware is available.
- If problems arise, the interface can be broken higher, above the site of the problem to enable continued testing while the problems are resolved.
- Testing is not as vulnerable to shuffling hardware configurations, especially if hardware must be pulled for repair or rework.

3.3.3 Integral Test Software Architecture

It is apparent that during test the boundary between flight and test software will become quite indistinct in many configurations. This is entirely appropriate, since the lines between reality and simulation fall upon the same boundaries as the layering of the overall architecture described earlier. Over the course of testing, as more of the system is put in place, this line gradually moves outward to the hardware, and then beyond to the physical interface between the hardware and its environment, later to be displaced entirely by the reality at launch. Until this last step there is a component of the support environment that is retained through each stage. Continuity should be retained throughout for those components spanning test stages with no need to change simulation models, support equipment, or other elements of the support system.

Likewise, the test support system must be an integral part of the entire software development process, not just a parallel effort that meets the flight software at some later stage in the development. The same incremental approach must be followed to make test capability available from the start, as needed by the flight development.

3.3.4 Inclusion of Operations in the Test Support System

To be a truly unified architecture for flight and ground, the test environment must transition seamlessly into flight with operations providing equivalent support to both arenas. This makes the operations software an effective subset of the test support system, so it must effectively support the system throughout development. In essence, operation software should become what is left of the support environment at launch. Given the layered architecture whereby portions of the operations capability may be shifted to flight (or conversely), it becomes doubly clear that the basis for distinguishing among flight, test, and operation software is vanishing with these areas comprising nothing more than a partitioning of components to different physical locations in an otherwise homogeneous software architecture. The recognition of this must become both technical and programmatic in order for a unified architecture to succeed.

3.4 Accommodate a Dynamic Development Environment

There is always a danger with any effort like the UFGA that it will become rapidly obsolete, serving more as a detriment to progress than a help. An ability to easily adapt and grow from mission to mission has been described. A similar flexibility is required, however, even within the scope of a single mission. As a system progresses from concept to implementation to flight many adjustments in the software approach will have to be made. Software is often left scurrying at the last minute to make the final accommodations.

To avoid this, the architecture must be nimble and lean. It must not be a overbearing, feature laden behemoth that moves only with arduous coaxing. Following are some of the situations that must be easily supported.

3.4.1 Late Scope Adjustments

Experience tells us that the scope of a software development effort can be quite dynamic.

Reductions

When well laid plans go awry, it may be necessary to exert a lower level of control than originally intended. This may happen during development if obstacles are encountered, or it may happen in flight if difficult problems force close intervention.

Should such a situation arise, the steps necessary to shift the level of control earthward should be safe and simple, essentially through a process of lifting upper layers of the architecture. This should expose no interface not thoroughly exercised during the incremental integration and test process. This implies that flight/ground communication can be inserted between any two layers.

Enhancements

Planning will occasionally not permit the development of full capability at an early stage. This could be a consequence of problems, or it could be a deliberate step to shift development efforts off the critical path into a quiet period of interplanetary cruise. On very long flights there may even be a notion that waiting is better for the program by allowing technology to progress further before development choices are made.

The steps necessary to enhance capability may include replacement of software components, or the shifting of control back to the flight system. Both should be supported in a graceful manner with any added capability exploiting existing flight interfaces to the extant capability.

3.4.2 Design Changes

Software changes frequently due to changes in hardware design or to refinements in our understanding of its operation and failure modes. This is harder to accommodate if there is no explicit representation of the hardware in the software design. If only the analytic byproducts of our hardware models reside in software, then every hardware design change forces us to revisit each conclusion, procedure, and test — a process that can never be thorough.

Model Based Design

Part of the solution to this problem is converging on the correct knowledge of hardware behavior sooner and more precisely than we have in past programs. Model based design is a key tool in addressing this motive and is rapidly gaining momentum as the design paradigm of future programs.

Model based design will allow hardware designers to address issues and find problems at an early stage by giving them an executable specification of the design that can be engaged with other models. Integration starts much sooner, giving software development the lead it needs and giving it an opportunity to remain current. The models also provide a less ambiguous truth test against which the behavior of actual hardware can be gauged. Moreover, it will be more difficult foist hardware idiosyncrasies upon the software as not strictly forbidden behavior, whereas any departure from a model will be viewed as exceptional.

Model Based Implementation

The same modeling effort used for design can have an immediate impact on the software design if the software is prepared to take models as direct input. One level of realization this might consist of automatic code generators working from a model of the desired component functionality. Mature tools of this sort are presently available.

In a more idealized conception, the flight software would be comprised of generic “engines” and models of the hardware and environment. All other behavior would be derived by inference, given the goals of the system. In reality, this cannot presently be fully achieved in a practical way, but with specialized representations of the models, supported by additional heuristics and guidance from designers, substantial strides have been made in this direction.

Advancement in model based design and model based software implementation are harmonious objectives which should be supported by the UFGA. More will be said later on this subject in the context of software reuse.

3.5 Define a Clear Evolutionary Path for Advancement

Each new program is tempted to start anew with every software development rather than building upon past efforts in a process of continuous refinement. Two of the obstacles that have stood in the way of this more rational approach are fundamental incompatibilities between present flight and ground architectures, and a uniform method of partitioning and design that makes functionality available in an easily reusable form. This has made it very challenging to evaluate the success and merits of a software design. A principled basis of software design evaluation is essential to analyze where specific improvements are necessary, and to find where opportunities for alternative solutions exist. The architecture principles required to gain these properties are described below.

3.5.1 Migration of Capability Between Ground and Flight

The trade between flight and ground implementation involves competing issues. There will always be more computing capability in ground systems than flight systems, and human interaction, when required, is most readily done with ground systems where communication constraints are minimal. On the other hand, there will always be more data available in the flight system. The data will be immediate, not delayed for hours or days, and reactions can be effected much more quickly. Data can also be processed on board to more effectively use the available link. Therefore, as more computing capacity becomes available in flight systems and the quality and reliability of flight autonomy improves, the balance shifts increasingly in favor of flight systems. In the extreme, all automation is on board and ground interaction is reduced to providing the subjective contribution that is irrevocably in the human domain.

This ultimate aim is far into the future, so in the meantime it is necessary to find a good partnership between flight and ground that fits the present state of development where substantial capability remains on the ground. Accordingly, it is likely that many of necessary flight elements eventually required will first take form in ground systems. This doesn't mean in all cases that proven ground capability can move intact to flight systems. Their unrivaled opportunity for immediate response will always give flight implementations a fundamentally different character from their ground counterparts. In fact, any capability, for which movement to flight exacts no significant alteration, could as well remain earthbound. Migration, therefore, is not simply about the movement of software.

What then does migration of capability from ground to flight mean? There are different answers, depending on the issue addressed by migration.

Assimilation

In the simplest examples, the desire is merely to break the tyranny of the link, which is expensive and time consuming. In cases where data is sent to the ground for automated processing, and results possibly returned to the flight system, but on a relatively leisurely schedule, the value of migration to flight is purely to save link costs. In this case, ground capability can migrate essentially unaltered, except as the need to fit more tightly

constrained computing resources dictates changes. An advantage may accrue from eliminating round trip light time in the reply, but the nature of the results is not materially altered in migration of this fashion, and it may well be that the saved time is used for nothing more than reducing the overhead of the task itself.

This sort of increase in autonomy, while not trivial, is not burdensome to implement, once the infrastructure for migration is in place. The enabling feature is an architecture that overtly recognizes the kinship of remote processes with local processes, in a sense making migration little more than the alteration of link characteristics between communicating software components. The accomplishment of this migration is not therefore so much a success for the capability moved to flight implementation as it is for the capacity of the flight system to assimilate it. Migration, first and foremost, is an architectural strength.

Augmentation

More complicated examples of migration deal not so much with transporting capability as they do with transforming capability. The function leaving the ground is reconstituted in a fundamentally different and more vigorous form on board purely because of the richness of data available that no ground rendition could command. Such functions are typically more intrusive and all-encompassing because they influence, not just *how* things get done, but *what* things are possible.

For an architecture to accommodate this sort of autonomy, the extant elements must do more than simply acknowledge their new partners. They may further be called upon to play new roles never required when greater control rested with the ground. For example, augmentations could be necessary in predictive or diagnostic abilities. Adapting to such immigrant capability, therefore, means also supplementing the capability already in place. The ease with which this can be done without undermining the inherited design is the second essential strength an architecture must possess to support migration.

Neither of these approaches to migration can happen by accident. The first has direct implications on the way architectural components interact and the independence of internal function from the external characteristics of interaction. The second suggests an approach to modularity wherein each component may be further partitioned into separable modules, residing in different locations.

3.5.2 Reusable Components

Having the design in hand for one mission, it is likely that the next involves most of the same functions, often with comparable hardware for much of the system. The next after that will similarly resemble its predecessor, and so on. One could conceivably find the subset of each design that continues into the next and rebuild the new design upon this foundation. After a few missions a thread that all share may be found winding through the collection, and there would be a natural inclination to broaden this thread to the extent possible so that each mission could minimize its development costs. This approach has, in fact, been in general favor for decades, describing much of the present infrastructure for flight projects.

The unfortunate side effects of this approach are that it either finds only the least common subset among all programs, or it severely constrains their implementation and retards progress. This can be partially mitigated by building software in a manner that is more

readily tailored for each application, such as so-called “data driven” implementations, or those that reuse a common kernel. Nevertheless, the collective strand remains a restrictive concept, addressing only broadly cutting functionality, joining a diverse spectrum of disciplines under one colorless cloud.

Self-contained Modules

Far less entangling is an approach that addresses the interoperability of diverse components. The essence of this approach is to encapsulate all aspects of a particular object within self-contained modules sharing common interaction standards, rather than drawing related functions from several groups into some monolithic program. In this way the identity of an object is not subsumed inside some self-perpetuating amorphous structure that has no allegiance to any particular system or component.

Such modules can move more freely and intact from one program to another, or can be set aside when not needed without threatening the integrity of other modules. There is no need to span most programs — only enough to make potential reuse attractive. Therefore, the overall portion of functionality captured in reusable form is potentially much higher. At the same time, each module survives only as long as it continues to serve a vital purpose, and this capability can be abandoned in relatively small decrements. The agility of this process assures that continual progress is not hampered.

In additional to modularity, the concept of a reusable component implies that part of the component remains fixed as the component is reused. However, it need not be all of the component. The nature of the reuse depends on what parts are held fixed. This can include each of the following cases:

Interface reuse

> Interface reuse occurs when the interface with which the component interacts with other components remains fixed while the internals of the component may vary. Interface reuse is commonplace in software libraries and protocols (e.g., math library functions implemented in hardware or software floating point arithmetic). While the implementation may vary, there is nevertheless an implied constancy in the function actually provided. That is, one’s expectation of the results of using the component may be considered part of the interface which remains unchanged.

Algorithm reuse

> Algorithm reuse occurs when the functionality of the component is reused in a variety of contexts, where each context requires a specific interface. With a complex data processing algorithm, for instance, there may be substantial benefit to reuse in a new system, even if substantial interfaces changes are required.

Component reuse

> With component reuse both the interfaces and algorithms of the component are reused without modification. Legacy software falls into this category where the software does not adapt to its environment but the environment adapts to work with the legacy software. One gains the greatest benefit from this type of reuse if the architecture defines standards capable of minimizing the level of adaptation required.

Note that the scope of component reuse can occur both within one software design, across multiple software designs, or both.

Adaptable Modules

There is still plenty of room to create adaptable modules that are not bound to a particular system element. A variety of methods can be applied, including those used in more cross-cutting approaches, but also model-based implementations as described above. This has the effect noted earlier of broadening the applicability of a module, but in a much more focused manner that permits deeper representations, thus improving the chances that a module can be applied to a given application.

3.5.3 Promoting Design for Reusability

Reusable software components require an architecture to promote reuse based on modularity and interfaces, but equally essential is a development process that makes software design for reuse a natural thing to do. Software development involves a series of steps from software requirements, through specification, design, implementation, and test. Various development strategies (e.g., waterfall, incremental, or spiral models) differ on the basis of granularity, scope, and revisions. These distinctions are orthogonal to the techniques for designing, implementing, and testing the software. Therefore, the architecture alone does not provide all the ingredients necessary to realize reusability.

Custom software development ranks low in that regard since reusability is not a primary concern of the software designer. Object-oriented development methodologies provide a plenitude of solutions, but also introduce the risk of compromising reusability due to methodologies and tools that others may not be able to reuse.

Model based software development offers an intermediate paradigm where reuse can occur not only at the level of the software product (i.e., algorithms, structure, and interfaces), but at the level of the software development process as well. Its contribution is to capture purpose and functionality into a model and to use this model for producing (in part or whole) the design, implementation, and testing of the software. Models explicitly representing the purpose of software components in terms of the responsibilities bestowed on them, and they describe how components are designed to achieve their intended purpose. Model based software tools then translate this purpose and design into operational products such as source code, documentation, command lists, telemetry catalogs, and interface headers. This has important consequences on the processes of software development and reuse.

In this manner, software reuse can occur in form independent combinations of:

- Using existing models or writing new ones
- Using existing model-based tools or writing new ones
- Targeting models and tools for the same software architecture or a new one

[See “Model Based Software Design” in Appendix B — Examples.]

Standard Interfaces, Structure, Etc.

A completely formless approach to modularity is inappropriate. Some issues are enduring, regardless of the vehicle, mission, or epoch. Basic notions of time, resources, reliability, priorities, and so on pervade most functions and should be addressed directly in the UFGA. Interoperability across a variety of interface characteristics, functional migration, incremental capability for assembly and variable fidelity execution, and so on also dictate a set of universal standards of implementation that must be established. This section has suggested many such attributes that can guide the architecture. The aim of the UFGA should be to put into effect these principles in a thorough, well-structured manner that solves recurring universal problems, while maintaining the flexibility of functionality necessary to gain wide acceptance.

o

4 An Approach to Layered Design

To meet the guideline of a layered architecture where increasing capability is incrementally available, and operational at successive stages, it is necessary to establish an approach to system hierarchy that identifies capability at each level. One such approach is described here.

In this discussion two types of layering are intended. At a coarse level there exists across the whole architecture the notion of named layers such that each layer has a well-defined interface, each interface supports a well-defined protocol, and each software component exists within a particular layer. This supports the goals stated above regarding incrementally available capability and layered operation. Without well defined layers these goals are more easily slighted. Remote interfaces will also impose clear delineations of function that are best matched to comparable divisions within the architecture hierarchy.

Within each of these layers functionality may be further “layered” in the sense that some components control others, but at this level of granularity, there is no need to further subdivide into strict layers. In this context, “layering” refers simply to the hierarchical organization of components.

4.1 Flight

This layered approach will be described beginning with the most basic flight systems; then showing how each layer builds upon those before without corrupting the preceding layers. At the most autonomous extreme, most functions reside in the flight system (or systems) with little but operator interfaces left on the ground. To present the most unified picture, layering will be discussed first in this context. However, it should be noted that the intent is to make the dividing line between flight and ground possible at each stage, retaining one or more upper layers in ground systems in very similar form to that which they would assume on a flight system. The additional discussion required to address the ground portion will therefore be minimal. Test systems, which in this architecture are essentially a combination of the eventual ground system plus a simulated flight environment, will be discussed last.

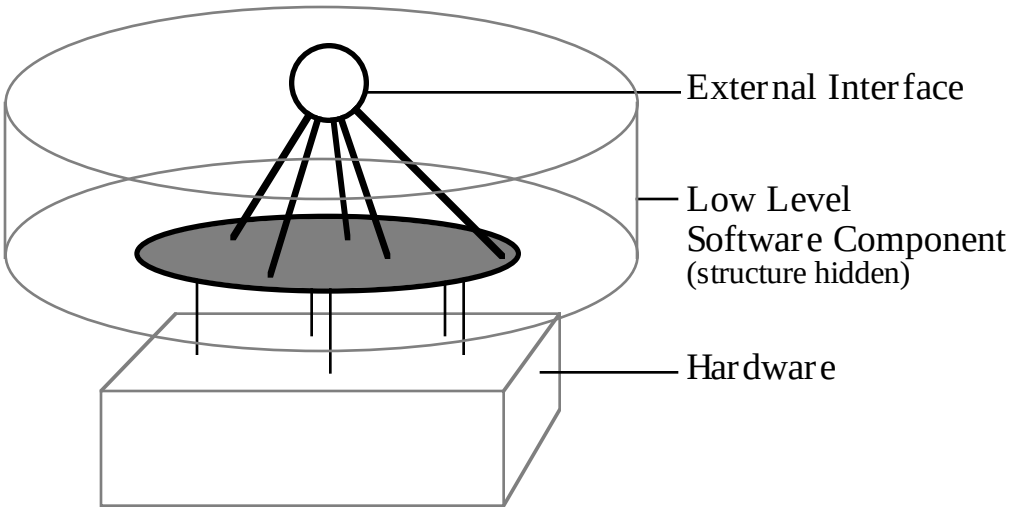
4.1.1 Basic Systems

Below a certain level, especially where computing resources are highly constrained, the overhead of a sophisticated architecture can become unreasonable if it too highly focused on the larger effort. However, the UFGA must not abandon smaller applications.

Instead, the nature of software components at the lowest levels of the UFGA architecture must be such that each addresses the comprehensive local needs of its associated hardware element.

That is, each low level component of the UFGA architecture must manage all aspects of a particular collection of hardware that collectively performs a highly integrated function. This will typically involve local maintenance and control of the hardware, often in a reflexive manner, in compliance to simple commanded goals and constraints. It will also involve reporting the results of these commands, including the status of the system and its ability to support the commands. And where goals cannot be met, there will be sufficient local actions taken to mask problems or at least preserve a safe configuration. A small number of operating modes will generally describe the whole component, all elements of the system working collectively to a single end.

The internal structure of these elementary components of the architecture is not overtly dictated by the architecture, except to the extent that it meets infrastructure constraints and subscribes to the external interfaces imposed by the architecture. Therefore, there is no point in discussing further hierarchical structure within the component.



For compact systems such as micro-rovers, surface penetrators, and the like, where extremely efficient implementations are demanded, this can be sufficient to meet all local needs. That is, for such systems, the “flight” portion of the UFGA may consist of only a single low level component. By addressing low level software components in this manner, a full UFGA implementation is still possible even for simple systems, since this lowest level component is a complete solution, and only this lowest level component need reside with the fielded system. This extends UFGA support to such systems, while allowing them

nevertheless to operate within the context of a larger implementation, the bulk of which resides elsewhere, such as on the ground or in a remote support vehicle.

The architecture here seems to be that at this level everything but the interface is outside the architecture. Explain.

In more complex systems several such elementary components may reside in the same system, each dealing with a different subset of the hardware. This situation is described in the next section.

4.1.2 Cooperative Interaction of Functions

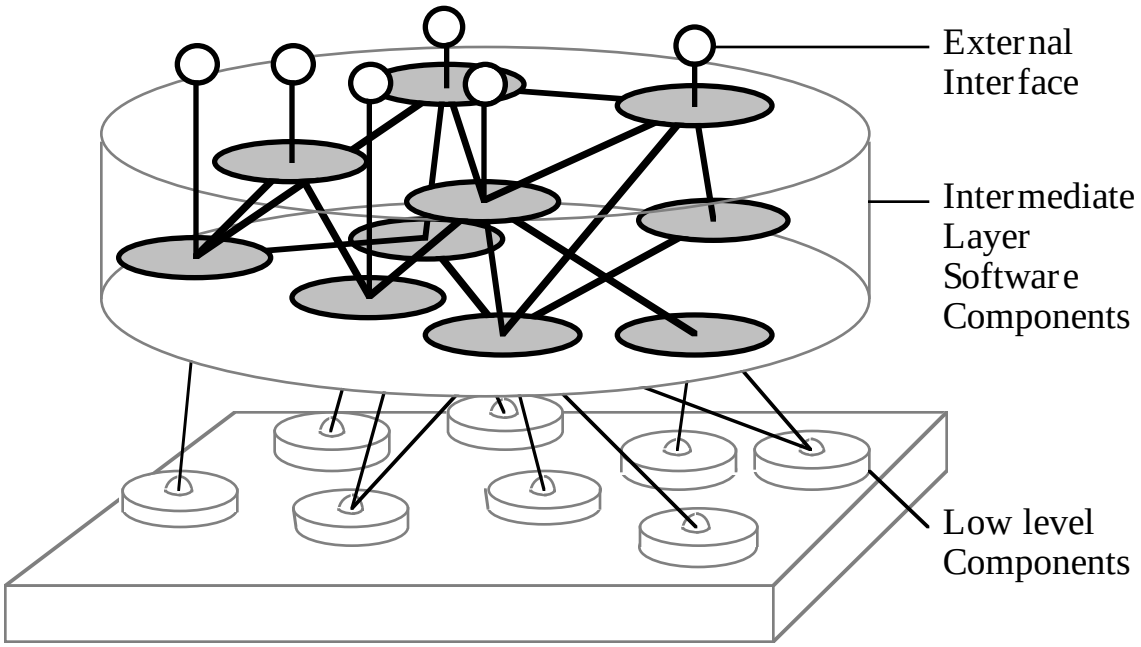
A larger system will be comprised of several low level functions working cooperatively on a larger task. These may be instruments, sensing or actuation assemblies, power supplies, or other components of this sort.

In this context, low level components can be associated with what are generally referred to as hardware managers, but it is important not to impose this interpretation too strictly. There will be situations where inherited designs bring with them pre-existing local software, which the rest of the UFGA must accommodate. By not imposing internal structure on low level components, the architecture is better suited to deal with such inevitabilities — even if it has to put a wrapper around such components to help them adhere to interface standards. After all, this is effectively the same function provided by hardware managers for their associated hardware.

Making each low level component of a larger system relatively self-sufficient, in the manner of the small independent systems described above, changes the collective architecture of the low level components only in regard to their physical affiliation and interaction, and the expediency of intercommunication this affords. Fortunately these changes are complementary.

This acquired need to address interrelationships and collective conduct introduces the next layer of the architecture — those components which build larger functions from lower level components by facilitating and managing their interaction in a productive way, adding additional processing if necessary, and then presenting this cooperative behavior through a single interface to higher levels. Coordinating sensors and actuators (via their low level managers) into a control system is an example of components at this level.

There is no attempt at this stage to consider elaborate goals coordinated over long time frames. Instead, this first intermediate layer is characterized by aggregate capability that mimics in many ways the nature of its single constituents. That is, the collective behavior too will include maintenance and control functions, meeting relatively immediate goals and constraints, reporting status and activity, reacting to problems and so on, in a fairly reflexive manner. The main difference is that the internal interactions contributing to behavior at this level are happening within the context of the unified architectural structure. That is, the method of interaction among components in this layer will be dictated by the architecture.



Beginning at this level, the interdependence of components may vary by function and circumstance such that the system becomes multi-modal, sometimes capable of supporting parallel activities or of configuring components in assorted combinations to different ends. More elaborate goal and constraint commanding methods become necessary and must be explicitly recognized within the architecture definition.

Each component of this layer will typically be dedicated to a particular set of related functional modes. Many will directly access low level objects, though not all need do so, working instead through intermediaries in the same layer. Therefore, some components in this layer will be subordinate to others. Generally between two linked components, if one component is ever subordinate to the other the reverse will never be true. Thus, while there is no further layering among these components, there is a partially ordered hierarchy among them.

Some pairs will be linked as peers, generally when both are subordinate to a common controlling component that supervises the link. Not all pairs are linked, nor must the topology be static. These relationships are established by the needs of each function as provided by or to other members.

It is possible, and probably common, that some components will be subordinate to two or more others. Thus the topology of the hierarchy need not be a tree. Nor should there be restrictions on the span of any such link.

Components may then be in competition, depending on the mix of present goals. Since this occurs at visible points within the architectural framework, mechanisms will become necessary within the architecture to manage the disputed resources and resolve stalemates. However, in a relatively steady state environment these allocations may still be handled in a reflexive manner with bounded expectations on the resulting behavior. This gives the source of commands a basis for planning activities over long time spans.

There is a question regarding the extent to which desires expressed earlier for incremental assembly and variable fidelity execution should be satisfied within this hierarchy.

Incremental assembly (from the bottom up) is answered by noting that any component high enough in the hierarchy to have no superior can be eliminated without impact as long as subordinate components have a defined, safe behavior in its absence. This is an important property all components must possess, not only for this purpose, but also for general robustness in the face of problems. Furthermore, this requirement propagates outward to the system hardware, which likewise must establish a safe state in the absence of supervision.

Variable fidelity execution can be addressed in various ways, including simply substituting simpler versions of all lower level components. There are situations, however, where this is not desirable — where one clearly wishes to address *only* higher level behavior. Goal directed commanding provides part of the answer, since its whole purpose is to hide the details of lower level operation, making goals happen despite disturbances. Provided there are no insurmountable resource conflicts, one may assume goals are met and therefore not bother to include the lower level components that perform the detailed operations. Realistically though, it isn't possible to ignore all potential conflicts. The best that can be done is to bound the behavior of missing components to some extent and retain at least enough of the lower level behavior to address the remaining issues. Each component should therefore possess bounded behavior under most circumstances, and be able to inform higher components of these bounds as part of establishing every goal.

4.1.3 Coordination in Time

While the goals instructing a moderately complex system can be elaborate, spanning long time intervals and parallel functions, the goal elements directed to components of the intermediate layer will necessarily remain at a moderate level due to their lack of any overt control over the system as a whole and inability to see far-reaching global consequences. The elaboration of system level goals and their coordination to achieve a common, overarching set of objectives requires some form of broad executive control.

One approach to this need is simply to continue building the previous layer in an ever deeper hierarchy until some component emerges at the top, serving as the interface for the entire system to the outside world. In keeping with the description for components of that layer, the type of control that results remains essentially reflexive, with high level goals elaborated through downward direction, advanced appropriately through feedback from below, and all activities coordinated to eliminate interference as the need arises.

At some stage in the hierarchy, though, this becomes limiting if goals at the highest level remain singular in time. Potential for parallelism in the lower levels is not fully exploited, and any notion of an agenda or preparation for future goals remains outside the system. That may be appropriate in some cases, and if so the portion of the architecture allocated to flight implementation may end with only the first two layers.

Otherwise, it becomes necessary to put some level of control over the timeline into the flight system. At the level where protracted management of time becomes a dominant aspect of a component's functionality, new issues begin to arise that are difficult to handle with purely reflexive methods. This is also the level at which the particulars of various system functions become sufficiently distant to enable a more generic control architecture.

Recognizing and providing solutions for these issues is the objective of the next layer in the architecture.

Time-based Sequencing

Time-based sequencing engines have been the traditional approach to dealing with time in past systems. Timed sequences provide the flight system with a superficial grasp of the future. If all goes according to plan then what should be done when is fully represented. Embellishments of this basic scheme can be elaborate, with hierarchically constructed sequences of sequences, "parallel" sequences, "parameterized" sequence constituents(i.e., macros), and so on. However, the axiom of predictability that enables time-based sequencing to work sets definite bounds on the range of systems that can use it.

The great success of this approach so far has been largely due to flying missions where the spacecraft either stays far away from a celestial body, or plants itself at one spot on its surface. Some types of uncertainty can be dealt with via goal oriented commanding, but without any feedback into the sequence itself, every such action must be bounded by worst case times and are local at best. Any remaining uncertainty that can't be dealt with in advance is generally handled by hunkering into a safe state while the ground regathers some understanding of the environment that allows it to get back into the prediction business. This has worked because we have done our best to avoid circumstances where it doesn't work, and because for the first few decades of space exploration there have been plenty of new targets fitting this restriction. This convenience can't last forever, though, and we are already beginning to see the end of it.

The limitations of time-based sequencing have become most painfully obvious in the past upon encountering uncertainty without the possibility of ground intervention. This is apparent, for example, when some critical undertaking like orbit insertion must be fashioned out a time-based sequencing system. Conditional waits and retries have often been the extent to which the core method could be augmented. Most of the actual effort to make this reliable has been accomplished outside the timed sequencing model.

These limitations are not cause to abandon this approach. They merely suggest that its scope of application is far from universal. Consequently, there should be room within the UFGA for it when appropriate, but other approaches must also be available.

Event-based Sequencing

The next set of enhancements to time-based sequencing generally considered allow the initiation of timed sequences to be governed by trigger events, or branches within them to be governed by system state. This can add a great deal of power and flexibility to the sequencing process, and provides a level of capability adequate to capture many missions where simple timed sequences would fail. Uncertain arrival times and distances on fly by missions can be accommodated within this model, for instance. Similar capability has also been used to implement system level fault protection.

With enough work a basic system of this sort could be used to do virtually anything imaginable. It provides all the capabilities of a procedural programming language, limited only by the access it has to system state, and the expressiveness of the conditions it is able to represent. In fact, procedural languages specifically suited for such applications have been developed. These provide a substantial improvement over the methods commonly in use today. Of course, it is still always possible, if appropriate, to revert completely to pure

time-based sequencing with such languages without having to resurrect an old sequencing engine.

The UFGA should also support such languages. Even better approaches are available, however.

Advanced "Sequencing" Languages

Despite the power of procedural sequencing languages, it can still takes a great deal of work to implement the sorts of goals and constraints one wishes to effect. This is especially true when attempting to organize multiple, competing, conditional threads of activity. Such situations are already common and will become the norm for ambitious future missions. Even when one believes a set of procedures is in place to handle the required goals and constraints, it is never clear from inspection that they truly capture one's intent. Testing must be exhaustive in order to build confidence, and there is little ensuing benefit from the development and test of one procedure that can be applied to the next.

Fortunately, while such procedures are difficult to create and verify, expression of the governing goals and constraints themselves is generally much more straightforward. Therefore, there is a great deal to be gained from a declarative style of "sequence" programming, which can leave the timely elaboration of goals and constraints into actions to an automated process.

This results in a number of important advantages:

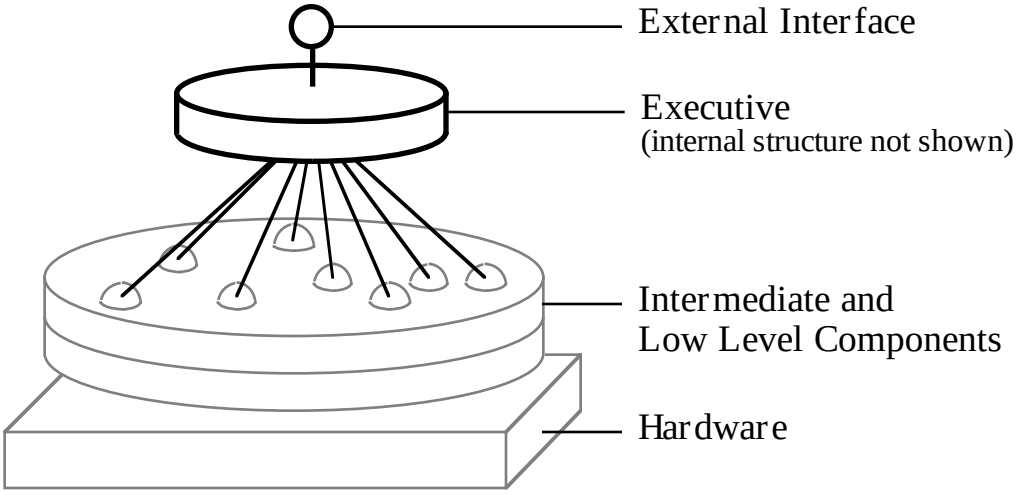
- Implementation of one's intent is directly verifiable.
- Priorities and alternative actions are easily expressed and incorporated into the process. This includes notions of which are expendable versus which are worth retrying.
- Much more complicated networks of temporally constrained conditions can be imposed upon a system than is practical otherwise.
- The system itself can tell you when it cannot meet its requirements.
- The engine behind the automation, where the investment is heaviest, can be reused in almost any context. Once verified, this certification carries over from one program to the next.
- Goals and constraints can be merged, dropped, or altered incrementally without having to rework issues of interaction. Giving up due to problems, or having to start over, is not necessary — a feature that can be crucial in critical situations.
- Changes due to faults, shifting environments, and other factors can be accommodated automatically and on-the-fly.
- The goals and constraints can themselves be the object of other programs, such as planners, making integration of such technologies easier.

It should be noted that this approach does not prevent the direct specification of timed or event driven activities — at any level. In fact, this highlights one more advantage of the declarative approach:

- A mixture of externally directed mission activities (e.g., science observations) and autonomously generated activities (e.g., orbit corrections) can be more easily intermingled, especially on short notice.

The Executive Layer

Which of these approaches is used has little effect on the resulting structure of any flight system that includes the executive layer. This layer consists of single component (or collection of components working together for a single aim) with interfaces to all visible components of the previous layer. The difference from one approach to the next lies in the degree of feedback exploited. Time-based sequencers use virtually no feedback, whereas more advanced approaches can potentially monitor a system down to a very low level of fidelity.



There is an obvious trade to be made between the level of distributed reflexive behavior permitted in the lower layers versus the degree of central control asserted from the executive layer. It must be stressed, however, that these are not competing issues. Rather, they are complementary issues that must be balanced in light of the circumstances for each mission. There is not one ideal mix for all situations. The UFGA must both accommodate this mix and allow for its tailoring to each application.

It should also be noted that the presence of "sequencing" capability in the executive layer by no means prevents reduced forms of it from being present in components in the lower layers. There are many needs and opportunities for sequencing in these components, even though they may be very limited in scope. Therefore, another feature of the UFGA should be to make such facilities (in probably simpler form) broadly available in lower layers, without having to replicate them everywhere they are needed.

Finally, it has been noted that each of the approaches presented supports a representation of the preceding simpler approaches. That means that any input understood with one approach should be understood, in concept, by each of the more capable approaches. Moreover, it is clear that some of the components in lower layers will likewise be communicating in similar terms that are conceptually just smaller renditions of the more capable executive. In the UFGA this idea should be moved beyond concept into reality by finding a universal protocol for the expression of such communication, whether single commands, simple sequences, or complex networks of activities.

Internal Structure

As noted, the executive layer need not consist of a single component. It may be internally complex with functionality divided among components of different specialties. It can be anticipated, however, that these components work among themselves in a much more conflict free manner than at lower levels. This is due to the nature of the executive as the ultimate resolver of conflict. If it has its own internal conflicts, there is no further appeal, so they must be dealt with internally. That isn't to say that all conflicts arising from lower levels must be addressed completely by the executive. It may in turn wish to appeal such conflicts to higher levels still (e.g., ground operators) for ultimate resolution. The point is merely that such unresolvable conflicts should not arise from within the executive itself.

This is not a trivial qualification. The difficulty of achieving such clarity can be seen, for example, when redundancy exists in the portion of a system which houses the executive. How does the executive decide when to move from one home to the next? Can an executive in so much trouble that it needs to move be trusted to make the decision in the first place, and then carry it out successfully? Having moved, how does it make sure no defective clone is left behind? Should there instead be two executives that try to agree with one another? How is a defective executive discovered? How is it terminated? Whose decision should that be? How is it possible to guarantee a correct choice? Should there instead be three executives with a majority vote? How complicated does this have to get?

Such questions get no simpler when redundancy is absent. They merely shift to issues of when the executive decides to step aside, letting lower level behavior take charge of system safety.

Issues of this sort make the executive layer more than just a high level extension of the intermediate hierarchy described above. The issues confronting it are unique, and investment in them can be large. Therefore, an executive is one component (or set of components) more profitable for reuse than most others, and consequently should be developed with great flexibility in mind.

4.1.4 Deliberation

The tasks performed by the executive are dictated by a "sequence", or more generally, by a set of goals, along with a description of the temporal and conditional relationships among them, their dependencies on external state, and information about relative priorities, alternatives, and constraints. The term "sequence" is taken below in the more general sense, despite the serial connotation of the term.

The creation of a sequence can be quite complicated. In order to devise rational behavior that will achieve the goals without violating any constraints, one must know about all the near term and longer term interactions among the goals, sub-activities, resource contentions, and so on. This type of global reasoning about how the spacecraft will achieve its goals is the objective of the deliberative layer of the architecture.

Planning and scheduling systems are a prime example of deliberative systems. These take a set of high level goals as input and produce a sequence of lower level activities that achieve the goals while satisfying operational, resource, and other constraints. This is a difficult problem, and a powerful reasoning engine is needed to automate this process. Depending on the efficiency of the deliberative system and the complexity of the reasoning problem

itself, it can take significant computational resources (minutes to hours) to produce a useful sequence.

At the other end of the spectrum is reactive or reflexive reasoning, which makes fast decisions based on local information. This kind of reasoning requires fewer computational resources than deliberation, but cannot reason about all the global implications of its actions.

Many intelligent agent architectures have both deliberative and reactive components. The deliberative portion is necessary to plan the longer term behavior of the agent, and the reactive portion is necessary to react quickly to unexpected or unpredictable environmental events. Without deliberation, the reactive system can make locally optimal decisions that make longer term goals unachievable. Without reactivity, the deliberative system cannot react quickly enough to external events.

The UFGA must therefore support both reactive and deliberative components. The executive layer and its subordinates provide the reactive functions in the UFGA. The deliberative system sets up a long term plan of activities which the reactive components carry out. By restricting reactive decisions to meet explicit assumptions and constraints in the plan, the reactive system can deal with unexpected events but still guarantee the validity of the plan. This prevents the reactive components from "painting themselves into a corner" while still allowing fast reactions to external events. Such boundedness must be a feature of the UFGA executive and lower layers.

Traditional flight operations can be considered a hybrid architecture, where the mission planning personnel are the deliberative system and the flight software is the reactive system. Simple engines, such as sequencers that do macro expansion, do not perform any deliberation directly, but rather invoke the considerable deliberative effort of human planners to coordinate all the complex interactions (much like block expansion is coordinated now in traditional sequencing). This requires much less computation, but much more human involvement. Therefore, the human component of operations cost is not fully addressed by this approach. Moreover, the timeliness of data accessible for such planning and scheduling can be quite poor. This is adequate to meet many situations, but due to its limitations, automation of the deliberative process is a key objective of the UFGA.

Initially, automated deliberative processes on the ground can work with a reactive system on board. The UFGA should support a scaleable level of autonomy, from detailed commanding at this level, to fuller autonomy where the automated deliberative and reactive systems in the flight system are given considerable decision making authority.

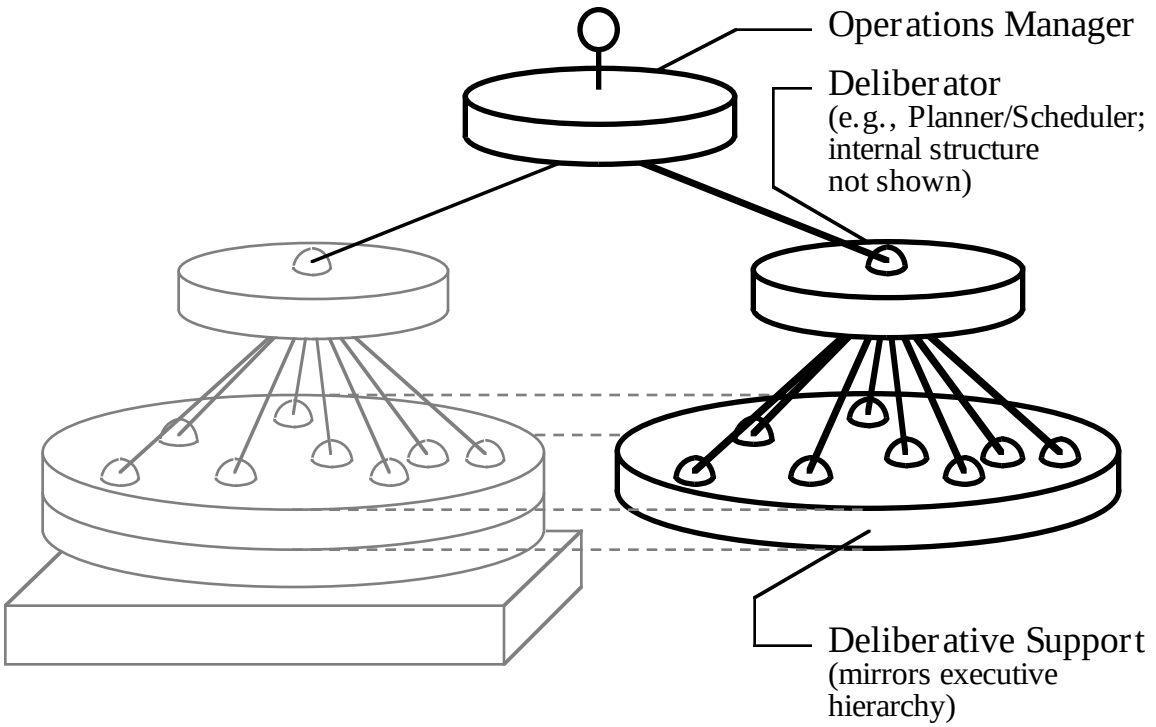
Defining the structure of the architecture at this level can be perplexing. It may seem at first that the executive layer is subordinate to the deliberative layer. This seems especially true when sequences come from the ground and the executive is the entity being commanded. On the other hand, sequences must occasionally be abandoned in the presence of unforeseen circumstances. In a fully autonomous system, the response could be to re-invoke planning and scheduling with new inputs on the nature of the new situation. Alternatively, the situation might be appealed to ground intervention. Either response is a reflexive action clearly initiated by the executive.

This dilemma is resolved by noting that both execution and planning/scheduling are subordinate to a higher authority. In the case where ground involvement is requested, humans will likely fill this role. In a fully autonomous system this role is filled by an

automated agent of the ground that is “aware” of mission objectives and priorities. Either way, both the reactive and deliberative components are at the disposal of this entity, which may be referred to as the operations manager.

Before considering the resulting structure, however, note that the deliberative system is not just a single component, nor need it be viewed as residing purely in a deliberative layer. Deliberation, after all, is not the solitary occupation it may seem. First of all, large amounts of data about the state of the system may be required to form intelligent plans. To be up to date, this information must come from components lower in the architecture. Second, the analytic abilities necessary to assess various system functions may lie (and should lie) within the lower level components of the system which perform them. For these reasons, any deliberative component requires links into lower level components to gather data and request services in support of the deliberative process.

Access points into the system should be adjacent to those exercised by the executive. That is, inquiries for deliberative support should go through the same hierarchy. Otherwise, it is possible for the eventual executors of a plan to find the plan inconsistent with their capabilities. This would violate basic modularity principles. The resulting structure looks like this.



Given this structure, it is apparent that reference to all the new deliberative capability as an additional layer is inappropriate. Only the operations manager lies in a new higher layer. The top level of the deliberative branch is a peer to the executive branch, and the components supporting deliberation are at the same level as corresponding components in the execution hierarchy. Most of the new structure is an augmentation to the structure already described.

Defining the structure in this way is important for one crucial reason. In particular, note that the entire deliberative hierarchy may reside on the ground. This describes the traditional ground sequencing process, as well as any advanced sequencing process where it has been decided, nevertheless, to retain deliberative action primarily on the ground. On the other hand, this structure also supports a fully autonomous system where deliberation is performed largely in the flight system. Thus this structure supports the notion of migration to flight all the way from traditional sequencing to full autonomy without fundamentally altering the architectural structure.

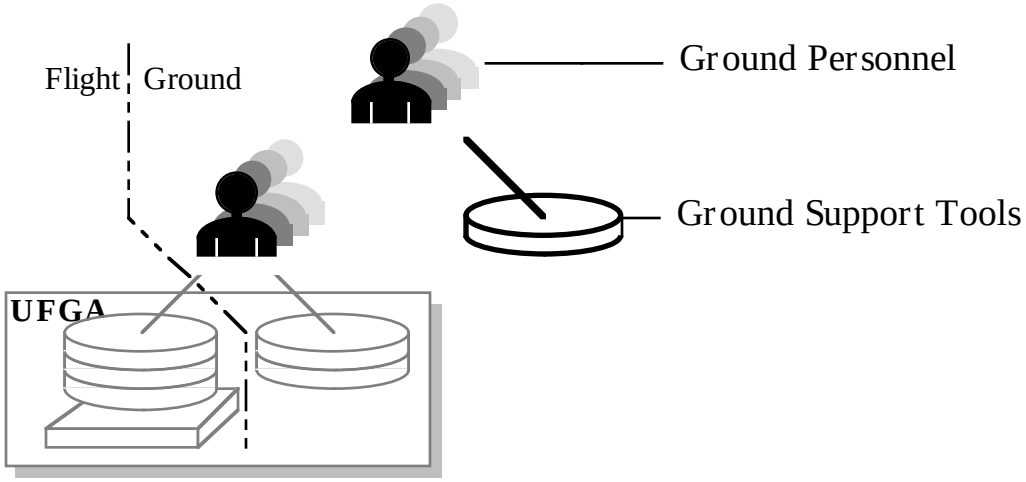
In this structure it is the operations manager that invokes planning activities. It is also the operations manager that commands the executive, either with sequences produced by the deliberative process, or directly, making it possible (if perhaps not advisable) to bypass the deliberative process. If the executive determines that a sequence can no longer be executed in its present form, it informs the operations manager of this problem in a completely analogous manner to the way subordinates inform the executive, and so on down the hierarchy. It is the operations manager's decision whether or not to invoke a replan — an essentially reflexive behavior! Whether it does so may depend on the criticality of the mission phase in juxtaposition to safety and long resource considerations. Therefore, the operations manager may be thought of as super-executive, operating reflexively from a very coarse grain sequence of mission goals, priorities, and constraints. Present day operations managers are teams of humans. Fully autonomous systems hand this responsibility to an intelligent agent in the flight system.

Mission Plans

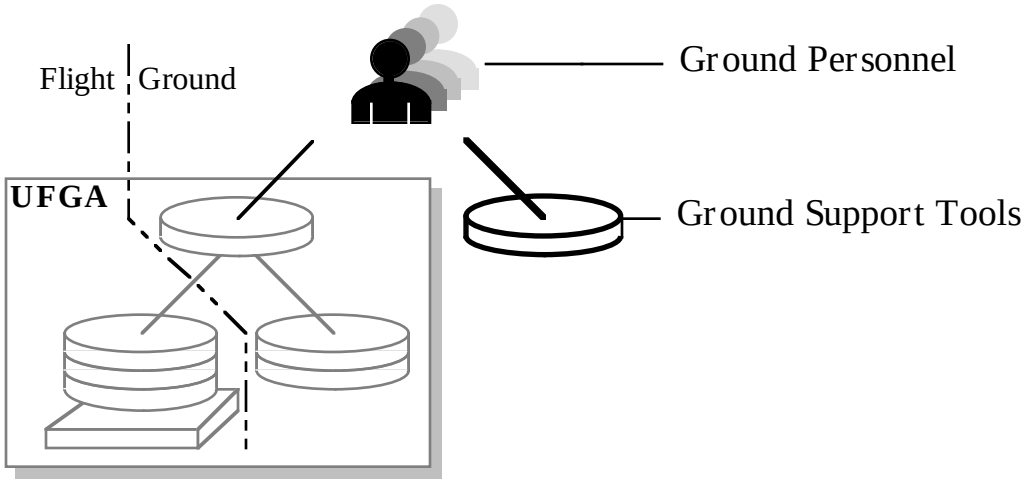
In this concept it is possible for systems at the autonomous end of the spectrum that mission plans move from broad allocations and guidelines made for human consumption to formal specifications made for machine consumption. As such these plans will assume a shape similar to the more detailed sequences discussed earlier. The essential differences will be that goals are longer term and higher level, dealing with major mission phases and events, priorities will deal with strategic issues such as safety goals versus broad mission goals, and constraints will deal with issues like long term resource consumption — sufficiently so that involved deliberative processes can be included in their methods of behavior.

This is also the level at which human involvement is irreplaceable. Therefore, the generation of mission plans will always be the domain of humans, no matter how automated the tools which support them.

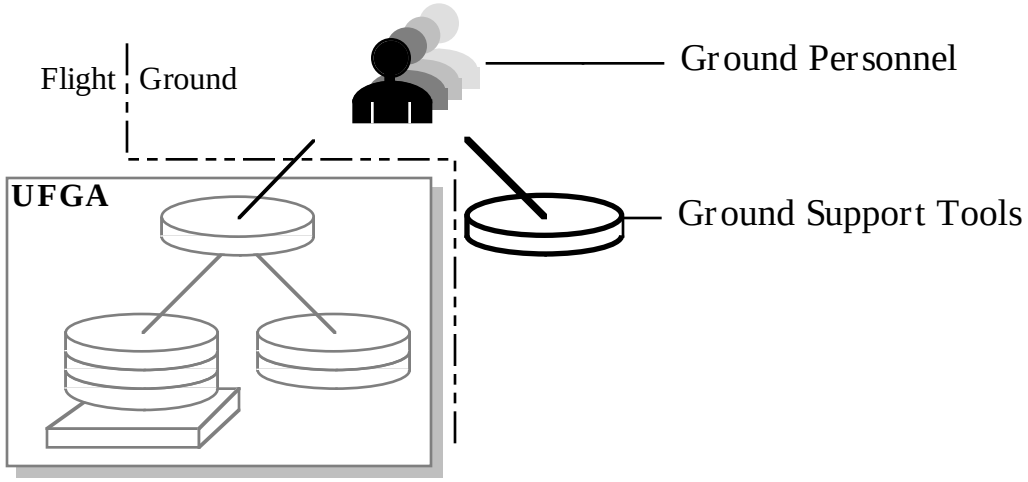
Traditional



Partial Automation



Fully Autonomous



Planning Granularity

Granularity of reasoning in the deliberative process, whether fully automated or not, should be dynamic, as dictated by the uncertainties in behavioral bounds in lower level functions for various situations and the need for optimality or long term constraint satisfaction. Where possible, it is generally advantageous to plan at the coarsest level, letting lower level processes expand high level goals into detailed actions. This is especially true when uncertainty is greatest and a highly reflexive approach is necessary to react to the situation as it unfolds. Critical events often have this character, but it also describe a typical day of in situ exploration. Coarse grain planning also applies, though, when the schedule is relatively relaxed, allowing reflexive processes ample freedom of action, even if the result isn't too efficient. This level of automation might handle large stretches of a typical mission.

On the other hand, when conditions are well known, but there is strong motive to optimize a sequence, planning to a finer level of granularity may be appropriate. Science observation sequences may fall into this category. Detailed planning may also be necessary in a highly constrained situation where purely reflexive processes are unlikely to find an adequate series of actions satisfying the constraints.

Most missions are likely to have a blend of these situations. Therefore, the level at which deliberation occurs should be adjustable as the mission progresses, for example as part of the mission plan.

Replanning

A potential problem with deliberative processes is the time they can take to generate a plan. With enough lead time this is not an issue. Working from the mission plan, it may be possible to start the next phase of planning hours or days in advance, since the end point of the previous stage is well defined. However, in an unanticipated situation that requires a prompt reaction, this is unsatisfactory. Reflexive actions may be able to deal with the immediate situation, but if a full response requires rapid regeneration of the sequence, there is a problem.

Traditional sequencing has partially dealt with this predicament by breaking so-called "critical" sequences into short restartable chunks and then highly engineering the responses to problems for that particular set of activities in an attempt to put the interrupted sequence segment back on track. In addition, contingency sequences are often prepared to deal with certain scenarios that can be anticipated and where staying on the original plan is not possible. These are limited methods of dealing with the problem, and so expensive that their use is confined to only the most severe circumstances. Problems that arise during more mundane activities generally result in dropping large sequences that can last weeks.

Besides emergencies there may be other motives for wanting to alter a sequence in progress. New constraints or new goals based on information that was not previously available are examples. Unless windows are left in sequences specifically to permit the late addition of new activities (an inefficient and not necessarily adequate approach), the only way to deal with such changes while leaving the sequence unchanged is to let reflexive processes sort it out. For the same reason that deliberative processes are needed in the first place, this is an unsatisfactory approach.

The automatic ability to replan addresses some these concerns, but if replanning takes hours, then hours of activity are lost. The extent to which this can be shortened determines the extent to which replanning becomes an effective tool both in contingency situations and in the normal adjustment of the mission plan. One approach to this problem is a deliberative process that is able to iterate from a previous plan, adapting just enough accommodate the new input. Another is a deliberative process that generates successively better plans starting from a crude but "correct" initial product. This way, it is able to provide the best available when needed rather than having to wait for the undiminished process to play out. Even when there is time to deliberate to an ultimate conclusion, the ability to pause part way to incorporate new input can have significant advantages. However it is accomplished, these flexibilities in the deliberative process should objectives of the UFGA.

Links between Reflexive to Deliberative Processes

In the structure described above a duality was suggested between each component in the execution branch and a counterpart in the deliberative branch. These doublets cannot exist in isolation from one another, even if the first resides in the flight system and the second in the ground system. The deliberative component shares information with its executive counterpart, and when flight conditions affecting it change, the deliberative component may need to know about it.

From an architectural point of view, this implies that there is a peer to peer link between such pairs of components, even if this link spans interplanetary space. The UFGA should explicitly provide for such needs.

Monitoring Services

Software components, and the hardware they serve, can be coupled in ways not overtly reflected in the hierarchy established by the architecture. In fact, part of the reason for reflexive behavior in the system is to justify a level of ignorance about such things. It may still be wise to monitor such interactions though, to guard against changes that violate assumptions. Furthermore, it is often the consequences of these overlooked interactions that provide the clinching evidence against a particular device as the origin of a fault. This means that some sort of monitoring service, looking for interactions not strictly within the avenues of normal component discourse may be necessary.

If for no other reason, basic telemetry will always be needed where unfettered access to data may mean direct access to each and every component, regardless of the functional hierarchy.

This does not imply that the structure as described so far has omitted something essential. It does suggest, however, the existence of components that are in some sense omniscient, needing access to many if not all other components. The UFGA should provide the means to handle such functions efficiently.

4.1.5 Distributed Systems

Have all layers of the architecture been identified? Not necessarily, but at this point it is possible to build upon the capability of existing layers, expanding functionality through nesting and parallelism.

For instance, note the similarity in structure between operations personnel providing mission plans to an intelligent agent with the help of ground tools to an automated operations manager providing sequences to an executive with the help of deliberative processes. The hierarchy within the execution branch can be similarly nested if two or more relatively independent systems comprise the executive's domain.

Suppose, for example, that an orbiter is in charge of a family of small rovers or aerobots. Each of the in situ systems could possess it own internal executive with subordinate components, but this collection of executives might be under the control of a supervising executive in the orbiter where most deliberation occurs. Similarly, independent spacecraft flying in formation might take direction from a common mission planner on the ground or in one of the vehicles.

Deliberation might likewise be distributed among specialized deliberators which generate fully formed plan constituents (under constraints) which are then incorporated into a larger, more general plan.

Differentiation and segregation of structures within the UFGA in this way should be the method by which the architecture extends itself over a diverse array of distributed configurations.

4.2 Ground

Much of the ground portion of the UFGA in this layered approach has already been revealed. From the mission plan through deliberation, sequencing, and execution, the ground plays a greater or lesser role depending on how much of the architecture is retained on the ground — capability moving a layer at a time. The architecture, as described, provides a broad spectrum of choices on this division.

Other issues regarding the flight ground split are less structural in nature. For example, the amount of on board data reduction that occurs can be decided almost invisibly within the context of the structure. This is an orthogonal issue that involves trades on bandwidth, storage, computational power, compression losses, and so on. The impact on the architecture has more to do with making sure such trades do in fact remain independent. Nothing about the architecture itself should impose a choice one way or the other.

Other issues to be discussed…

Data bases and archiving

4.3 Test

o

5 Software Structure

A software architecture may be described both in terms of the general structure it imposes upon a software design without regard to any particular application, as well as the more particular functional organization imposed by the application. This section deals with features that transcend application issues. Functional issues are discussed in a subsequent section.

As a general architecture, to be applied over a progression of applications, the UFGA requires a software structure that permits inheritance and migration of capability, with the consequent desire for uniformly implemented modules that can be composed in different ways. In addition, the interactions imposed by shared processing, communication, and other infrastructure, the hierarchy of a layered organization, the mutual high standards for reliability, testability, and other attributes, all motivate norms of implementation across components of the architecture. Even within a single application, a harmonious software structure is important in generating a manageable partitioning of the effort that can be easily integrated.

While these statements may apply to virtually any system and there are many competing approaches to their realization, there are nevertheless common obstacles faced by systems for remote exploration that recommend a more narrowly focused selection of characteristics. Moreover, an approach appropriate at one level in the software structure may be impossible at another level, especially in a system where separate elements may be anywhere from occupying the same computer to occupying different planets. Characteristics deemed advantageous to meeting these needs are addressed below.

5.1 Object-oriented Modularity

An object oriented approach to software [See “Object-Oriented Software” in Appendix A — Definitions] creates a modular partitioning of a design that improves independence of development activities, and potential for migration and re-use. It is also well suited for weakly coupled multicomputer architectures (often preferred for reliable, fault tolerant designs), and for widely distributed systems inherent in remote exploration. Moreover, it more cleanly divides into replaceable or optional partitions in support of incremental delivery and layered execution levels.

5.1.1 Hybrid Approach

Due to the distributed nature of the UFGA architecture, even in its humblest incarnations, a system will generally consist of several active objects. The flight and ground components of the software are clearly distinct in this way, but many other physical divisions may force this. This does certainly not mean that all objects within the architecture must be active objects.

At lower levels within the architecture, there are compelling reasons for several objects to share the same thread of execution. Synchronous I/O and control functions operating cyclically with precise timing requirements may find the overhead and sampling jitter intolerable, especially on low performance or highly loaded computers. While the resulting level of coupling is tighter, it is to a desirable end that is difficult to achieve efficiently in other ways.

At intermediate levels, interactions tend to be more asynchronous and infrequent. Making objects active, even where not dictated by physical partitioning, is an effective way of decoupling objects.

These observations indicate that the UFGA must support a combination of active and passive objects on the same processor. This would presumably by accomplished within the multitasking capabilities of an operating system wherein some objects would share a single thread of execution maintained by the operating system or by one parent within the group.

5.1.2 Object Interactions

To the extent possible, objects should be able to interact with one another in a uniform manner across a wide variety of circumstances. At one extreme, one object can invoke another through a simple subroutine call. This would be the case, for instance, when low level objects share a process, one calling the other.

At the other extreme, an object on a computer on the ground can invoke an object on an embedded processor on a spacecraft through a complex series of events that includes multiple DSN passes.

It is desirable to make the programming of inter-object interactions as uniform as possible at both extremes while recognizing that the runtime behavior of these interactions will necessarily be very different. To accomplish this it is necessary to have an abstract description of inter-object interactions that captures the salient inherent differences imposed by physics and the design of digital systems.

Attributes

We identify five dimensions along which to describe these interactions at an abstract level as follows.

> **Latency** — How long does it take between the time an object is invoked to the time when the invocation has its intended effect or returns its intended result?
>
> **Reliability** — What is the probability that an object invocation will fail to have its intended effect or produce its intended result?

Side-effects — Is the object invocation designed to compute a result or to produce side-effects such as changing the value of a state variable or taking some physical action? Side-effecting objects can require synchronization which non-side-effecting objects do not.

Parameter passing — Are parameters passed by value or by reference? If by reference, how are object interactions handled across multiple processors and multiple memory systems, and how is memory management accomplished? If by value, how are aggregate and linked data structures handled? (Needs clarification)

Blocking — Does object invocation cause the invoking object's thread of control to block? If not, by what control mechanism are results returned to the invoker?

Inter-process Interactions

Active objects (i.e. processes) interact with each other in more complex ways than passive objects. Passive objects, because they do not possess their own control thread, can only have their methods called by other objects as subroutines. Active objects, by contrast, have their own control thread and thus interact in more complex ways. For example, active objects can potentially exist on multiple physical processors, possibly using multiple memory systems. The following issues need to be addressed by the inter-process interaction mechanism.

Mutual exclusion — Some mechanism must be provided to mediate access to shared physical and logical resources like shared memory, mass storage devices, or spacecraft actuators. Simply assigning each resource to its own "manager" object does not solve the problem because the manager object may be invoked by multiple threads. It is possible to solve the problem by putting critical sections entirely inside single method bodies, which assures mutual exclusion if the manager object is single-threaded. However, this approach only works if mutual exclusion extends over short periods of time.

Preservation of argument semantics — When objects reside on heterogeneous processors some mechanism must be provided to translate data between the formats used on those disparate machines. There are two ways to do this: marshaling, and putting semantic information in the message content. Both approaches have tradeoffs in terms of flexibility and computational costs. Also, many marshaling implementations imposed an additional burden on the programmer because they cannot directly parse data structure format information from the source language, and require the programmer to supply redundant information in a different format. This in turn requires either manual maintenance of these redundant representations, or the use of additional development tools to manage the redundant representations automatically.

Contingency management — In multiprocessor systems, the integrity of the computational infrastructure cannot be assumed. (In fact, the whole point of many multiprocessor systems is to provide redundancy in the event of failures.) The inter-process communications architecture should be robust in the face of both transient and permanent failures.

Control flow — How are multiple access to a single object to be mediated? Are access processed in a strictly FIFO manner, or is there some prioritization scheme?

Can one object cause an asynchronous change in the control flow of another object? If so, what are the restrictions and constraints? If not, how are time-outs implemented?

Abstract interface — It is possible to design an abstract interface to an inter-process interaction mechanism that hides various aspects of the underlying implementation, including: the host processor for any particular object, the architecture of the host processor, the communications latency between objects, etc. Although information hiding is generally good programming practice, in the case of a widely distributed system it might be desirable to make certain aspects of the implementation manifest in the API. For example, it might be desirable for an object invocation that relies on data communications between spacecraft and ground to look different from one that is a purely local computation in order to make it obvious that a high-cost operation is being performed.

Physical Interactions

In a fully object-oriented design there is no interaction among objects except through the methods presented to other objects. This extreme is hopelessly idealistic for embedded applications such as spacecraft control where software plays an interactive role with hardware. The state of a hardware manager object, for instance, may be viewed as the aggregate of its internal variables and the state of the hardware that it manages, since the object consults the state of the hardware as part of many method invocations. The physical portion of this state, however, also evolves through the invocation of physical interactions. Through this path the state of one object may influence the state of another without the invocation of software methods.

Physical interactions may be due to tight margins on shared resources or simply due to the interconnectedness of physical processes. Some of the more important interaction mechanisms are:

Kinematic state, particularly spacecraft attitude, but also including trajectory and articulations. Nearly every activity on a spacecraft imposes some constraints on spacecraft attitude.

Subsystem states that cross subsystem boundaries. For example, sending a command to an instrument may require a data bus to be in a particular state.

Renewable resources, such as electrical power, data storage, and communications bandwidth.

Non-renewable resources, such as propellant and operating lifetime of some hardware.

Electromagnetic, thermal, mechanical, and chemical interactions, such as EMI, vibrations, propellant sloshing, and chemical residues from engines and thrusters

Many interactions operate across multiple mechanisms. For example, on a solar-powered spacecraft, changing the spacecraft's attitude can change both the available power and the thermal load.

The above interaction mechanisms are examples of what is known as physical causality, the notion that there is an inherent ordering among phenomena due to the physical nature of the processes that are responsible for such phenomena. While there are several approaches to understanding and modeling causality at a level adequate for designing and building software components, model-based reasoning techniques offer several of techniques for modeling causality at various levels, and for separating the part of the causality that is inherent to the physical world from that which is induced by interactions of software components with each other and with the world.

Other Interactions

Software objects on the same processor also interact outside the strictures of method invocation through the limiting effects of bounded computer performance. At the least, one may view the present context as part of the shared state of each object. More importantly, one object cannot become active without another becoming inactive. Therefore, whether or not an object meets a deadline can depend on the operation of other objects

In addition, there are indirect interactions brought about purely by logical inter-object competition that can lead to deadlocks or other failures.

In either case, an object confronting a failure must consider an alteration of state to deal with the consequences, even though no method invocation has occurred.

Finally, to the extent that object oriented design is not followed, objects may interact through other means outside of method invocation.

5.2 Inter-object Communication Standards

Points to make:

Different approaches appropriate at different levels. Low levels need low overhead, low latency, high bandwidth. Intermediate levels need robustness and ease of integration. Remote communication links need to deal with long time delays and to support processes to improve reliability, and link efficiency.

Despite differences, there should be uniform standards of some sort. For example, migrating tasks, switching the link, and so on should be transparent at some level.

[The following makes several controversial points that need to be cleared up.

- *Most present inter-object communication uses RPC or client/server model*
  - *Only if discussion regards interprocessor or intertask communication. Between objects in the same task ordinary procedure invocation is still the norm. Many objects sharing the same thread of execution will remain an important part of future systems, as it is the dominant form now, so procedure calls must be included in this discussion.*
- *Blocking effectively reduces the number of threads running:*
  - *Only one thread at a time runs on one processor anyway, so this is only an issue for interprocessor communication. We've always used asynchronous messaging for that.*

- *Only the calling thread is blocked, but other threads on the same processor can continue to run, so the processor need ever be idle, even with blocked threads.*
- *There is often a natural sequence of execution among objects such that objects are blocked anyway, simply because they're waiting for the next round of processing.*

- *Synchronous communication introduces a class of deadlocks*
  - *This is true only if the blocked object is not reentrant, supporting only one thread at a time. This is generally a resolvable situation, but there are effective ways to avoid it in the first place, for example by restricting procedure calls upward into the hierarchy.*
  - *Objects can be internally multithreaded. This is often the only basis for resuming execution after sending a message anyway when the message expects a response.*
- *Asynchronous communication makes this class of deadlock impossible*
  - *The same potential for deadlock exists with asynchronous communication if a reply is required to proceed and the object blocks without it. That is, the communication system itself may be asynchronous, but objects can add their own synchronization requirements and reintroduce all the same behaviors as synchronous communication.*
- *All concurrent threads of computation to continue while data communication proceeds.*
  - *Only true if they have something useful to do while waiting for a reply (often not the case), and only useful if concurrent threads are on different processors so more than one can run.*

]

The UFGA will be a distributed system, encompassing multiple concurrent active objects or tasks running on (potentially) multiple processors. No such set of objects can function as a coordinated system unless they can exchange data. The means chosen to enable this data exchange will profoundly affect the character of the UFGA.

Most of the inter-object communication mechanisms in widespread use in the 1990s are based on remote procedure calls (RPCs) or remote method invocation, also known as the "client/server" model. This model leverages off software developers' familiarity with the notion of procedure invocation. In concept, an RPC is simply the calling of a function that resides in a remote address space; as such it can have side effects and/or return a value like any other function call. The intent is to enable the developer to develop distributed applications as easily as monolithic ones.

But when a function is called, the invoking (or "client") code is suspended until the function has completed and the result (if any) is returned to it. Deviating from this operational structure would devalue the RPC model by reducing the cognitive leverage it gives developers. Adhering to it, though, means that an RPC must similarly suspend — block — client code until the remote procedure has been completed and the result returned. That is, the client/server model limits the number of objects that can actually be computing (i.e., not suspended) at the same time, and the severity of this limit increases as the time cost of data communication increases.

Moreover, this "synchronous" communication introduces the potential for a class of software deadlocks that are subtly different from the resource allocation deadlocks discussed earlier, in that the computing objects themselves are the resources for which there are conflicting demands. Suppose object A issues an RPC to object B, and in order to respond to that RPC object B must issue a further RPC to object C. Both A and B are suspended, waiting for C, so neither of them can respond to RPCs issued by other objects until C responds. At least three threads of processing have been effectively reduced to one, that of object C, but that's not the only danger. Suppose at some point the implementation of C is changed such that it must obtain information from A in order to respond to any RPC from B. There is nothing in C's source code to warn of the looming disaster, but as soon as this change is made and A issues its first RPC to B, all three objects — and all other objects that communicate with any of them, and all others that communicate with those, etc. — freeze solid.

Asynchronous message passing, in which objects merely send messages to one another and continue immediately without waiting for replies, makes this class of deadlock impossible and enables all concurrent threads of computation to continue while data communication proceeds. (See the discussion of the Law of Demeter elsewhere for additional thoughts on this topic.) For these reasons, asynchronous message passing is superior to the RPC model even if its utilization is less obvious to developers.

Basing inter-object communication on asynchronous message passing has other advantages that are particularly relevant to the X2000 program. By enabling client software to continue operating while data are in transit, it simplifies the implementation of communication between a spacecraft software object and one that resides in a Mission Support Area on Earth: communication over the deep space link takes longer than communication within the flight processor, but the very same application programming interface can be used in both cases without affecting the behavior of any individual object. That is, within some limits, the implementation of the object that sends a message can be entirely independent of the location of the object that receives it. (Tight control loops can't operate across the space link, of course. However, the delivery of observations to a planner and of plans to an executive might not be so severely affected by imposing or removing long round-trip light time delays.) This in turn simplifies the migration of functionality between the spacecraft and the ground, an important feature of the UFGA.

Finally, asynchronous message passing greatly simplifies — perhaps enables — the implementation of highly fault-tolerant software architectures based on parallel software object populations that "vote" at key points in the flow of processing. By thus reducing the need for radiation-hardening of flight computers, this technology offers the potential for significantly reduced spacecraft cost.

5.2.1 Layered Hierarchy

A wealth of interaction mechanisms make it more difficult to design layered hierarchical control for systems like spacecraft than for systems where the components are more loosely coupled. For example, a conventional computer system has a layered control system with device drivers at the bottom, operating systems in the middle, and application programs on top. But this is only possible because the writers of device drivers can safely assume that changing the internal state of a plug-in expansion card in a computer will not, say, cause an adjacent card to overheat. Such assumptions often cannot be made on a spacecraft.

There are two ways to address this problem. The first is to design a collaboration mechanism by which objects negotiate with one another to insure that their interactions are properly managed. The other is to have a supervisory control mechanisms that manage the interactions. In practice, both techniques are necessary.

Nested Layers of Control

["This section seems to say that control decisions can be made locally within objects. That is, a goal object can manage the interactions of the sub-goals beneath it, decide locally whether to abandon a subgoal, etc. However, this assumes there are no interactions among goals, or between a goal and the subgoals of some other goal. In general, these interactions occur all the time.

I think this approach will work well for execution, given a high-level plan (or program or sequence) that addresses the interactions. It may also be able to solve execution-time problems, if the solutions are guaranteed to have local effects and not violate the assumptions of the high-level plan. However, there will eventually be problems that cannot be solved locally. For example, if a thruster is stuck off, this will increase turn time and may impact achievable deadbands and fuel consumption. This has implications for all of the current and future activities requiring spacecraft attitude. The solution may require fairly extensive juggling of those activities, which may in turn impact other activities in the plan. A deliberative system is needed to reason about these implications and resolve the interactions. Local repair strategies alone are not enough.

The object-based approach presented places strong implementation constraints on that deliberative system that may not be effective or efficient." — Smith]

I agree that local strategies are not sufficient. The idea is to have constraints on behavior flow down along with the commands so that each object knows whether its local actions are creating global conflict. These constraints would have to be generated by a deliberative system and basically consist of the assumptions made by it on reasonable bounds of operation. If an object cannot honor the constraints placed on it, this information would flow back up until dealt with at a higher level, including reinvocation of planning. I will try to clarify this.

Supervisory control does not mean a single point of control for a whole system. In the first place, this not practical for distributed systems. Furthermore, its implementation would simply create additional coupling mechanisms, compounding the original problem. Instead, consistent with the layered approach desired for this architecture, control from each control site is delegated downward to one or more lower objects working cooperatively. This implies several properties of the control hierarchy which must be formally incorporated into the UFGA architecture.

> **Goals** — Direction to lower level objects is necessarily at a higher level than the resulting actions taken by these objects. This generally means more than simple deterministic elaboration of this direction, where in simple cases this amounts to little more than a form of data compression on the interface. It is intended, rather, that the direction be in the form of goals which are attempted in the presence of potential uncertainty, including uncertainty in the details of interaction among participating objects.

Constraints — Each object must accept, in anticipation of directed goals, a set of constraints on the actions it may use to achieve these goals. Constraints include both limits on resources and deadlines (constraints on the resource of elapsed time). These may be either implicit in the goals themselves or invariant properties of the object, or they may be separately imposed by the controlling object. Either way though, in order for an object to make commitments upwards in the presence of constraints, it must carry corresponding authority to impose appropriate constraints downward.

Bounds — Each object should also expect a set of bounds on the goals requested of it. Again, these may be either implicit in the goals themselves or invariant properties of the object, or they may be separately guaranteed by the controlling object. Either way, it may be assumed by an object in making its commitments that subsequent goals will be consistent with these bounds.

Commitments — Each object must have the ability to declare in advance of accepting a set of goals whether it is capable of accomplishing these goals under the imposed constraints and obligated bounds. Having affirmed such a request, an object must sustain the commitment until the it is released by the requesting object. While the commitment is held, the controlling object is free to issue goals consistent with the bounds and constraints of the original request.

Such commitments need not be hard guarantees, but they should be of sufficient confidence to prevent thrashing as a result of too many commitments that are made but subsequently denied. The flexibility gained through the potential softness of commitments is a key ingredient to attaining the loose coupling desired in the architecture. Note especially, that when high latencies are involved, commitments may need to be largely implicit, with the commanding object accepting a higher risk that goals may not be fulfilled. Alternatively, commitments may need to be established far in advance with reservations of required resources. Either approach requires some form of planning to anticipate and coordinate needs

Priority — To manage the degree of commitment, where guarantees are not possible, a basis for judging the relative merit of competing requests must be available. Therefore, requests must be accompanied by a statement of priorities. It is conceivable that commitments might be in a nested form wherein successive subsets might be given increasing priority.

Abandonment — An object must have the ability to gracefully abandon a goal (or the commitment to a goal). This could be driven by a conflict that has been resolved unfavorably through priority arbitration, or through the loss of a required resource (possible by similar means), or due to a detected fault or some other unanticipated change that cannot be adequately addressed at the local level. Abandonment requires notification to the requesting object. There should be no circumstances in which goal failure goes without notification.

Arbitration — An object must be able to resolve competing requests locally rather than simply referring them to a higher central control point for resolution. This is not required or possible in all cases, but is highly desirable. It may be accomplished by serializing goals, descoping or temporarily suspending goals in some pre-established acceptable manner, or abandoning commitments — all in conformance to the established constraints and commitment priorities.

Resources — Making a commitment generally requires two things. First, the goal and constraints must not conflict with previously accepted commitments in a manner that cannot be resolved. Second, the object must be able to gain access to all necessary resources. This may involve obtaining similar commitments from lower level objects required to support the goal.

Note in this arrangement that parallel paths of control through the hierarchy are allowed where some objects may be servicing more than one controlling object higher in the hierarchy. Furthermore, there is no explicit requirement for a single superior authority over all objects. Whether or not this is useful depends on the degree to which local conflict resolution can manage all plausible situations. As necessary, central control is required to anticipate and coordinate interactions such that conflicts either do not arise or arise only in locally manageable ways.

Peer to Peer Interactions

The control hierarchy described above does not cover all relationships and potential conflicts between objects. This is often because two objects share in a task equally with no clear precedence between them, and yet may need to communicate with each other or cooperatively use a common resource which they do not explicitly control.

It may also be because the interaction is inalterably outside the span of the control hierarchy. This is particularly true of physical interactions between objects as discussed above.

However, even when not strictly necessary, it may be desirable to deliberately avoid hierarchical control for various other reasons, relying more upon reflexive interactions to produce an acceptable emergent behavior. This might be desired, for instance, when the level of commitment that objects can make is already weak due to environmental factors, or in critical situations when issues of fault tolerance may take precedence over performance. Such peer to peer interactions lead to a looser degree of coupling that can make the system more tolerant to unexpected conditions. Tolerance to failure in the vertical control structure, as described above, supports this approach.

When two objects, neither of which is subordinate to the other, communicate with one another in order to perform their required tasks or through side effects of operation, then by definition the means by which each is assured that the other is in a state necessary to support the interaction cannot be directly via hierarchical control mechanisms. Even when a controlling object coordinates peer objects to some extent, it may be difficult to synchronize the transitions of two objects into compatible states. Mechanisms for peer to peer interactions must therefore be tolerant of occasional incompatibilities and disturbances.

Peer to peer interactions may be active, where one object (the consumer) requests action of another (the producer) and expects a response, or passive, where producer's output is unsolicited. One way to handle problems that are known to be temporary is to mask the consequences.

From the producer's side of an interaction, this can be done by

- extrapolating over a gap in support,
- queuing requests for later response,

- performing an alternate action that is tolerable for a short time,

or by any other action that satisfies the consumer's immediate needs. From the consumer's side of an *active* interaction, this can be done by

- suspending a request temporarily,
- retrying a failed request,
- dropping the request and extrapolating from the last available data,
- selecting an acceptable alternate action, or
- abandoning near term low priority goals.

In a passive interaction where the input is desired, the only choice a consumer has is to consider the age of the data. The consumer may therefore temporarily mask a problem by

- extrapolating from the last available data,
- selecting an acceptable alternate action, or
- abandoning near term low priority goals.

When passive interactions are undesirable, the affected object must adjust its behavior against the disturbance to compensate for it.

All of these actions are local responses within the scope of a single object. In some cases, it is necessary for the consumer to be aware of a producer's problem before it can take appropriate action. Therefore, the UFGA must formally support the following mechanisms in peer to peer communication.

> **Acknowledgment** — In an active interaction, a producer must either explicitly accept a request or reject it within the constraint of a deadline.
>
> **Time tagging** — In an passive interaction with potential latency problems, a producer must mark each output with the epoch of its production. This may also be required in an active interaction if the requested data is not necessarily fresh.

No other formal control mechanisms are required.

Longer term incompatibilities must be avoided, but in such cases it is necessary to invoke the hierarchical control structure described above, either through appeal to a lower level shared object that serves as an arbitrator of the conflicted resource (the existence of mutually compatible states), or by direct coordination from a higher level object which obtains commitments for a mutually compatible state from each object before proceeding.

Resource Management (original cut)

[There are two additional cuts at this section below. These must be reconciled and then merged.]

A recurring theme in this discussion has been the notion of resources. In general, a resource can be any entity with an internal state that is subject to competing demands. In the case of physical resources, examples of state are the amount of available power or energy,

the amount of remaining propellant, the amount of free space on a mass storage device, the attitude or rate of a spacecraft, or the operating mode of a peripheral. In each such case, a single object should manage the resource, the state of the resource effectively becoming an appendage of the object state, either through direct measurement or through tracking models of the resource. The object becomes a proxy for the physical resource.

All requirements for a physical resource should be requested via the associated object through the hierarchical control structure. Upon making a commitment on behalf of the resource, the object could be granting either partial or exclusive access, depending on the nature of the resource. Subsequent operations on that resource need not, however, be made through the methods of that object, but could instead be side effects of another object operating on a physical device.

For example, a commitment for power could be made by the object representing the power resource to another object representing an instrument, while the actual use of the power is a consequence of actions on the instrument by the other object. These objects interact at both software and physical levels, the software interaction through the control hierarchy, while the physical interaction is a peer to peer interaction and occurs outside the software architecture altogether. What makes this acceptable is that the software interaction anticipates and bounds the physical interaction. This is the essential role of a resource manager.

Resources can also be purely software objects with no physical attachments. Examples are objects which perform algorithmic services (such as control laws) or perform coordinating tasks (such as sequencing a maneuver). In these cases, the object is often capable of supporting only one of several mutually exclusive activities at a time. Furthermore, these activities often span considerable stretches of time where the notion of interleaving parallel activities has little meaning. Therefore, determining the activity in effect becomes a resource issue and it is necessary for objects requiring such a service to invoke the same controls used for managing physical resources.

In the case of pure software objects, all operations on the state of the object can be made through the methods of the object, so there need be no interactions outside of object oriented interactions. Nevertheless, they may occur for various practical reasons and the same management strategy can be brought to bear.

Resource Management (proposed cut)

["Somewhat contradicts existing text." — Smith]

Various spacecraft activities compete for resources, and these contentions must be arbitrated. Because resource arbitration decisions can have far-reaching global impact on other activities, resource arbitration requires a deliberative system that can see all of the potential interactions and has the power to reschedule or preempt activities to resolve interactions. There are usually many resource conflicts to be solved, and resolving one conflict may worsen the conflict in another area (e.g., moving activities to resolve a contention over spacecraft attitude may create oversubscription of power). Planning and scheduling systems are designed for this kind of deliberation. Reactive, execution time arbitration based on local information will generally be insufficient.

For example, preempting power from the catbed heater may appear to be a good decision based on local information, but disastrous in the long term. There may be an upcoming

critical RCS maneuver that requires the catbed heaters to have been on continuously for the preceding thirty minutes. Since the heater was turned off, the RCS maneuver is postponed. However, this maneuver was needed for orienting the spacecraft for taking a science image. The science image can be substituted for a later infra-red image, but that requires warming up the IR camera now, which will pull more power. This kind of reasoning requires a deliberative system that can look at all of the goals, constraints, resource requirements, etc., and find a globally consistent solution. Trying to solve this problem at the local level will not work.

However, if the effects of the resource arbitration decisions are guaranteed to be local, either by the deliberative component or by system design, then local resource arbitration can be done. In general, these guarantees cannot be made for all arbitration decisions, so an architecture that only supports local arbitration will not suffice.

The architecture must support a deliberative component for resource arbitration, whether this be an automated planner (ground or onboard) or a human planner. It should also support reactive components for local arbitration in those cases where it is appropriate.

Unpredictable Resource Usage

Resources that cannot be predicted until execution time are a good example of where local, reactive resource arbitration can be used. For example, it is difficult to predict when heaters will come on and off, though it is fairly simple to determine at execution time when a heater needs to be turned on. The deliberative system can allocate some amount of power for use by the heaters. A local, reactive manager determines how this power is allocated to the heaters. If some heaters must be on or off for global reasons, these constraints are communicated to the manager by the deliberative system.

If there is enough time between when the system knows the heater needs to come on, and the time when it must actually be turned on, the deliberative system can perform the resource allocation task. Local repairs are made to the plan starting a few minutes ahead, and the plan continues execution without interruption. The advantage of having the deliberative system do the allocation is that it can reason about global impacts if there are any.

Resource Arbitration Methods

[Needs a small edit to remove redundancies with earlier material.]

Arbitration of conflicting demands for transient but exclusive control of shared resources is a problem in any non-monolithic control system. The UFGA must support a variety of arbitration strategies at various levels of resource granularity.

In the simplest case, the resource in demand is useful in isolation — that is, delivery of that resource alone is sufficient to enable successful continuation of an element of software execution. By simply queuing demands for the resource and locating service of this queue in a single resource management agent (that is not itself a source of resource demand conflicts) we eliminate conflict.

When some set of resources are useful only in combination, the possibility of deadlock arises: task A has locked resource X and will release it only after some doing something that requires that resource Y also be locked, while task B has similarly already locked Y

and can't release it until it has locked X. The problem of deadlock can be solved either by preventing deadlocks or by detecting and breaking them when they occur.

The traditional strategy for preventing deadlock is to require all resource users to lock resources in the same order (in the example above, modify task B to lock X before locking Y, just as task A does). This approach works, but it requires that all software developers agree on a canonical order for resource locking and scrupulously adhere to that order. As such it is highly vulnerable to communication failure among developers and to program modifications that subtly change processing order and thus inadvertently introduce violations of the resource locking protocol.

An alternative is to reduce each complex resource allocation problem to a simple one, i.e., handle each set of resources that must be locked in combination as a single virtual resource and dispatch demands for that resource from a single queue. This strategy is somewhat more robust, but it still relies on disciplined software design: every virtual resource must be identified as such, and new virtual resources may emerge as development proceeds.

Prevention of deadlock is desirable for reasons of simplicity and efficiency, but a capability for detecting and terminating deadlocks will always be needed for recovery of spacecraft functionality in the event of a deadlock prevention failure. Algorithms for automatic deadlock resolution have been a research topic for many years; they can be expensive to develop and operate. At the other extreme, deadlocks can be detected by mission operators and terminated by power cycling the flight computer. In general, the cost effectiveness of automatic deadlock resolution (which is to say, the right level of investment to make in it) varies directly with the incidence of deadlock due to failure in deadlock prevention. As in object-oriented programming itself, good development practices pay for themselves.

5.2.2 Standard Methods

Points to make:

> Control hierarchy, declaration of goals, constraints, etc., peer to peer resolution mechanisms, resource management, and many other issues mentioned above and in later sections should be formally required as standard methods on all objects.
>
> The applied standard may be a function of level, (e.g., more efficient implementation at lower layers), but should be uniformly applied at the abstract level.

5.3 Object Domains

In addition to the normal bundling of data and methods associated with object oriented design the UFGA must address the bundling of multiple related objects into domains. The pertinent relationship among these objects is attachment to a common body of knowledge regarding a particular device, environment, or other entity. For example, there might be a domain for a camera with one object of the domain serving as the hardware manager, another performing calculations for required exposure times at a target, and a third producing simulated images for testing, and a fourth displaying images to an operator.

There are several reasons for not simply merging all such objects into one. Most importantly, they may be needed in different places at different times, as in the camera example cited above. Some objects of a domain may reside on the flight computer; others in one or more ground computers. Some objects may be needed earlier than others, or may be discarded after test.

Even when together on the same computer, though, these objects may support several essentially unrelated threads of execution. For instance, some may be involved in real time operations while others support long term planning. It may also be desirable to capture some aspect of system interaction spanning many domains in a uniform way best handled through separate objects. For example, each domain involving a device that consumes power may be required to supply a power model object for that device in some standard form. Merging diverse functions in one domain into a single object, where not otherwise required, would create unnecessary run time coupling. Maintaining and developing each object separately may also be much easier, especially if there are motives for using different languages for different objects.

Why bother then to bind these objects in any way? The reason is the common roots from which many of the implementation details in each object arise. These can include, for example, parameters, models, and state information. To a large extent, many of these features can be resolved when the software is built. There is more to this issue, though, than merely drawing implementation details from a common data base, or grouping such objects in a composite delivery. During test or flight it is quite common to adjust such information as a result of operating experience, and it is often necessary to do this during software execution. Furthermore, it is generally desirable to make such changes consistently over all effected objects. In the camera example, changing a sensitivity parameter can require compatible changes to planned exposure times, to real time control settings, to and simulation and display tools. Overt recognition of such run time coupling is a neglected but very important issue that has created much operational difficulty on past missions. The UFGA should finally address this issue in a rigorous manner.

5.4 “Real Time” Execution

Much of any system targeted by the UFGA must be regarded as a “real time” system. This means that actions are subject not only to logical requirements, but to temporal ones as well. At some level this applies to all systems. There arrives a moment when postponing any further degrades the utility of the result beyond acceptability. When the limit pushed is merely one’s patience, nearly anything is tolerable. One learns to temper ones expectations to avoid disappointment. When physical processes are involved, however, the picture can become dire.

A great deal of effort has been expended historically in eking every ounce of performance from embedded computing systems to address this concern. With faster computers seemingly lifting much of this burden, the allure of comparatively simple architectures which do not explicitly address this issue has occasionally succeeded in pushing real time design techniques into the background. It would be a fatal mistake to let this happen in the UFGA. Matters of latency and deadlines permeate everything over time scales from months to milliseconds. Wherever time matters, the UFGA must explicitly represent the essential issues.

One such issue has already been mentioned in discussing communication between objects where replies are expected. In the UFGA each communication should have the option to carry a deadline, beyond which it may be assumed that the communication has failed.

Another issue was the imposition of constraints on goals, where one such constraint might be a completion time. The UFGA should support the specification of such constraints and provide mechanisms to make commitment to such deadlines feasible.

Other time-related issues that has been mentioned are the advance reservation of resources, and the duration of commitments. Again, both matters should be explicitly addressed within the UFGA.

Other real time matters to be addressed must include the following:

5.4.1 Efficient Cyclic Task Support

At the most basic level in any system are sampled data implementations of data collectors, filters, profile generators, control laws, and many other algorithms running cyclically in a fairly repeatable order. When the designers of these algorithms express their preferences, they almost invariably wish the timing to be perfectly periodic with no latency between input and output. While this is never realizable, except in approximation, the necessary imperfections can be maintained within manageable limits if purposeful steps are taken to accommodate this requirement.

This has often been accomplished in the past with a simple cyclic executive. This is efficient, but far too restrictive for most modern systems. Cyclic processes have also occasionally fashioned from a multi-tasking operating system, but support is generally limited to reading the clock, assigning task priorities, and scheduling timed events. Additional capabilities have had to be added at the application level where solutions are likely to be awkward or unsatisfying. Such capabilities include the following:

> **Multi-Rate Scheduling** — Not all cyclic processes will want or need the same cycle period. It should be possible to schedule multiple cycles. While it may be advantageous to use harmonically related periods, this should not be a restriction since occasionally a period is dictated by an external process, such as a rotation period.
>
> **Phase Locking** — When independent systems interact on a cyclic basis, each driven by an internal clock, it is usually desirable to phase lock the systems together such that execution between exchanges remain synchronized. At the most elementary level this simply amounts to agreeing on what time it is, an end that can be surprisingly difficult to accomplish — even between computers — let alone over interplanetary distances.
>
> **Load Leveling** — When multiple cycles are in operation in the same processor, the computational load can occasionally peak severely as cycles come into alignment. An overload can result can be missed deadlines, lost or colliding events, and other problems. This can be partially mitigated by shifting the relative phases of cycles, either statically or dynamically as circumstances permit. Automatic support for this process can be very beneficial.

Load Shedding — When preemptive actions fail to avoid oversubscribing computing resources, it is often better to perform alternate, reduced overhead processing for a cycle or to skip complete cycles than it is to fall behind and loose phase synchronization with other cyclic functions. Such actions, however, can result in lost synchronization events, missing steps in a sequenced process, and other problems. There should be explicit mechanisms for taking such steps that assure that potential problems will either be handled gracefully or at least recognized as requiring more serious action.

Time Tagging — When shifting loads, asynchronous events, and other variables in the time line alter data sampling from an ideal schedule, the next resort is usually to know at least when a sample was taken. This allows consecutive samples to be correlated more accurately, sorting out the effects of sampling jitter from data noise. Time sampling may have to be extremely precise.

Synchronous Sample Construction — It is often necessary to consolidate data from asynchronous samples into a meaningful aggregate that represents a single moment in time. Moreover, it may be useful to hide the effects of sampling jitter and extrapolate such virtual samples onto ideal periodic time marks. The same process can work in reverse, where computed outputs, ideally put into effect precisely on some ideal periodic schedule must be adjusted to match the actual time at which the action can occur.

5.4.2 Events Driven Tasks

5.4.3 I/O

5.5 Fault Protection (Function Preservation)

[This is a partial submission. It needs to be edited to fit into the outline and then completed.]

A "fault" is a physical or logical defect in a component/subsystem. An "error" is an apparent symptom/manifestation of a fault. A "failure" is the consequent inability to perform as designed.

5.5.1 What is fault protection?

The role of fault protection (FP) is to prevent failures, either by masking faults or by detection and response to errors. Fault protection decomposes into three phases: detection of failure symptoms, fault diagnosis, and failure recovery.

Symptom Detection monitors sensor information to detect any evidence of a failure. The complexity of this detection process depends on the diagnostic value of sensor data, on the observability of the system behavior and on the qualitative discriminability of the possible behaviors relative to nominal and failure cases.

Fault diagnosis is the process of explaining a failure condition to the occurrence of a fault. Several factors affect the complexity of diagnosis. Diagnosis is greatly simplified when a failure condition is symptomatic of a fault in a component. Diagnosis becomes significantly more complex when several faults can result in the same failure condition. Active probing can be necessary to discriminate among several ambiguous diagnoses.

Failure recovery depends on the available repair commands that the software component can apply on a faulty component such as hard and soft resets, retrying the operation, or switching to a degraded mode of operation. When all possible recoveries cannot eliminate the failure condition, the recovery of last resort is to declare the component failed and unavailable for operation. Recovery can be more complex when active probing is necessary to evaluate the success of a recovery action, when the outcome of a recovery is indirectly measurable, or when other actions are necessary to reconfigure the failed component or other resources for recovery purposes.

5.5.2 Architecture for Component-Level Fault Protection

In keeping with "Capability in Layered Increments" (sec. 3.1.3), fault protection is a responsibility that is necessarily divided among different layers of software components, with very localized device FP at the lowest level and system FP at the highest level. Thus, a software component is characterized by the limited set of responsibilities assigned to it. In terms of fault-protection, the limited responsibility of a software component is bounded by the set of local failures in its devices and sub-components occurring while executing commands or performing its assigned responsibilities. The fault-protection architecture describes how non-local failures are coordinated between a component and its parent.

Key elements of the local FP architecture

- *The top-level components in the hierarchy are the different ground teams operating the S/C (i.e., engineering, science, payloads) which have specific areas of responsibilities assigned to them. The engineering component has several sub-components corresponding to ground operations and to the overall system-level operation of the S/C. The on-board system-level component has the responsibility of handling all failures that have to be resolved on-board but that no other on-board component has a specific responsibility for.*
- *Within a component, FP must treat the observations it receives as trustworthy; it cannot make conclusions about the reliability of observations outside of its scope — that's the fault-protection responsibility of a higher level component.*
- *For a given component, the absence of input data and the inability of obtaining input data are two distinct events that may prevent the component of fulfilling its responsibilities in part or in whole. When such cases occur, the component has a responsibility of notifying its superior component that external conditions prevent it from operating properly. To the extent possible, standard mechanisms need to be used to refer to unknown inputs (i.e., can get data but there is none), unavailable inputs (i.e., cannot get data, whatever it is), under-range and out-of-range inputs (i.e., input data is outside the range of allowed inputs for this component) and nominal inputs (i.e., input data is normal in face value).*
- *Within a layer, the following information must be propagated to a higher layer when non-local fault protection is necessary: local failure symptoms detected, faults diagnosed and local recoveries attempted. Failures must be propagated so that recovery/reconfiguration may be attempted at a higher level. Faults must be*

propagated so that even if recoverable within the layer via a redundant unit, the degraded reliability is known.

Fault protection mechanisms at design-time and at run-time

Since software components will come from multiple sources, it is expected that there will be some variability in terms of how fault protection is addressed among components. Specifically, there can be no built-in mechanism to report, at run-time, information about the specific recoveries that may have been attempted, the nominal/abnormal status of input data and the changes of responsibility due to failures. For such components, it may be necessary to either embed such a component into wrapper to provide the appropriate fault protection interfaces with the higher-level component or incorporate the missing information right into the higher-level component.

To the extent that information about local fault-protection behavior will be shared among components at run-time, the architecture places requirements on how this sharing will take place. Fault-protection information exchanged between components includes:

From local component to upper-level component:

This information includes models used at the lower-level for diagnosis/recovery and the nominal/abnormal state of the input data used for local fault protection. This communication needs to proceed on the basis of a publish/subscribe mechanism for several reasons. First, the component hierarchy is not static since components can be migrated between ground and flight processors and among processors in a distributed computing network. To accommodate reconfiguration of the component hierarchy, it is necessary that indirect interfaces such as a publish/subscribe mechanism be used among components to reconfigure inter-component communication routing as needed.

From upper-level component to lower-level component:

Although a lower-level component may be unable to handle a non-local failure condition, the upper-level component should not take over the overall fault protection responsibility of both. Instead, the role of the upper-level fault protection component is to reconfigure the lower-level component as part of the recovery process. Component reconfigurations include:

- *turning off a component because the functionality it requires is no longer adequate for the component to meet its responsibilities,*
- *degrading a component responsibility to accommodate resource degradation outside nominal range.,*
- *redirecting component inputs and outputs to use redundant information sources and command outlets in order to avoid inoperable resources.*

5.5.3 Strategy

From a system engineering point of view, the role of fault protection is to prevent the inability of a component to meet its responsibilities. Tolerable failures include those that a component can recover locally without a visible impact at the level of the component interface.

Strategy

Focus on Function

Tolerability versus Performance

5.5.4 Local FDIR

Detection

"Parity" Tests (Sparse space of legal states)

Redundancy Based

Model Based

Lower level "Events"

Peer Level Masking

Restoration, Automatic Degradation, or Safing

Notification

Failure Events

Status of Remaining Capability

Requests for Diagnostic Activity

Event Logging

5.5.5 Escalation through Hierarchy

Layered Detection for Coverage

Measured Response

Reactive if possible

Replanning if necessary

Ground Role

“Not sure where this should go, maybe under fault protection, but the notion of "physical degradation" should be addressed as an issue for the software to be aware of and handle. It is related to periodic calibration.” — Dvorak

5.6 Operation

5.6.1 Startup and Shutdown

5.6.2 Maintenance

Module Replacement & Modification

Parameter Updates

Model Updates

Code Updates

New Objects

Calibration and Checkout

“Explain that there should be a post-launch calibration and checkout phase during which the autonomy dial is set at minimum and lots of telemetry is sent to ground to let humans examine operation in flight and look for stupid mistakes, like the sign of a gyro being reversed, or other things that weren't testable in flight system testbed. This is especially important before turning up the dial on autonomy.” — Dvorak

5.6.3 Selective Enabling of Control Layers

“To allow ground operations to gradually turn over control to the onboard autonomy. During this phase the ground should be allowed to observe what the autonomous response would have been had it been enabled; this will give them confidence in the onboard s/w.” — Dvorak

5.7 User Interfaces

5.8 Test

5.8.1 Simulation

Spacecraft

Spacecraft Trajectory

Spacecraft Dynamics

Rigid and Flexible Bodies

Fuel Slosh

Rotating or Articulating Elements

Small Forces

Sensor Input

Actuation

Loads

Forces and Torques

Thermal

Sources

Solar, Radioactive, Propulsion, Friction

Measurements

Thermal Effects

e.g., focal length

Telecommunication Links

Environment

Celestial

Stars

Celestial Body Ephemeris

Body Features

Rings, etc.

Radiation and Particles

Surface Maps

Forces and Torques

Gravitation and Gravity Gradients

Atmospheres (density, winds, …)

Contact Models

Radiation

Emulation of missing elements

Hardware

Software

Direct Access Interface

External

Substitution

5.8.2 Monitoring

Interfaces

Hardware Layers

Software Layers

Diagnostic Tools

Detection

Analysis

Visualization

o

6 Functional Areas

“There is a document on the JPL web titled "Spacecraft Information Function Taxonomy" by Sandy Krasner, located at: http://fst.jpl.nasa.gov/jpl/function.html. This memo provides a checklist of areas for which requirements must be defined for any spacecraft, so it may offer a fuller outline for section 6 "Functional Areas".” — Dvorak

6.1 Commanding

6.1.1 Goal Directed Behavior

[This is as submitted. There is a little conflict with other sections, and part of the material might find a better home in other sections.]

Goal Based Commanding

The spacecraft should support commanding at various levels of detail, from high-level goals to low-level commands. This allows the mission operations team fine control over the level of autonomy granted to the spacecraft at any point in the mission.

Each goal expands into subgoals at the levels below it, and manages resources and interactions among those subgoals as well as interactions with peer goals. By commanding at the top-most level, the mission controllers can set high-level objectives for the spacecraft which the autonomy system then achieves without further direction from the ground. To exercise more control, the ground team can command lower level goals. The spacecraft only has autonomous control over how those low level goals are carried out, and the ground team controls the sequencing and interaction of those lower level goals. Commanding at the lowest level is analogous to a traditional sequence.

It should also be possible to command at mixed levels. For example, ground can uplink several high level goals, but specify in detail how one subgoal is to be achieved. The spacecraft carries out the high level goals autonomously, but follows the ground directions in carrying out the subgoal. This allows the ground very fine grain control over the degree of autonomy granted to the spacecraft. The autonomy system must be aware of both the high-level and low-level commands, so that the expansion of high-level goals does not conflict with low-level directives.

The architecture should also support mixed-level commanding. That is, the spacecraft could be given a few high-level goals, but also be told exactly how to carry out one of the sub-goals. This capability is necessary for carrying out normal operations with high-level goals while performing specialized commanding at a lower level. For instance, the ground may wish to execute a high-speed turn to jar loose a partially deployed solar panel, but otherwise continue normal operations. High level goals are given for the normal operations, with a special subgoal for the high-speed turn.

Goal based commanding should support prioritized goals as well as absolute goals. A prioritized goal is carried out if possible, but can be preempted if it conflicts with a higher priority goal. Priorities can be either static or dynamic. In a static priority scheme, goals have a fixed priority ordering. The system finds a schedule that optimizes some function of the goal priorities— e.g., all of the priority one goals must be in the schedule before any priority two goal can be scheduled, or assign each goal a priority value and maximize the total value of the goals in the schedule. In a dynamic priority scheme, the priority of a goal depends on what other goals have been scheduled. For example, getting the first science image of a target has very high priority, but getting the hundredth image has very low priority; or the priority of downlinking a particular data buffer increases as the buffer fills and as the number of downlink passes in which it was passed over for other buffers grows.

Goal based commanding requires coordination among the goals, resource allocation and arbitration, and resolution of interactions among activities in the same or separate subgoals. The interactions can be complex, often involving several subgoals and resources. Resolving them requires a deliberative system that has visibility into all of the interactions and has the authority to reschedule activities and resources in order to resolve them. Automated planning/scheduling systems are ideal for this task.

Simple macro expansion will generally be insufficient, since the way in which the goals are expanded depends on interactions with other scheduled goals and activities. Distributing coordination efforts among goal, resource and activity "objects" will generally not work either unless there is some global coordinating entity that has visibility into all of the objects. However, visibility into objects breaks the opaqueness policy of object-oriented design, and the coordinating entity would effectively be a planner. This is roughly equivalent to organizing the planning model into goals, subgoals, and activity "objects" or "modules" that contain declarative descriptions of the resource, temporal and other constraints. The planning engine then has visibility into the various modules and authority to change the schedule within certain constraints (such as directives from ground on how to achieve a subgoal) in order to resolve conflicts.

The goals eventually expand into executable activities. That expansion should be robust to execution failures. The low level activities should themselves be goal oriented, in that they have some set of resources under their control and a well defined goal to carry out. The software unit for the activity can decide how to achieve that objective within the restrictions set by the deliberative system in expanding all the goals for the spacecraft. If failures occur during execution, the activity manager should first try to resolve the failure on its own, again within the restrictions of the deliberative system. These restrictions must be met to prevent the plan from being invalidated. If it cannot resolve the failure, or must violate the restrictions to do so, the failure should be passed up to the next level.

Ideally, the same behavior is repeated at the next level. That is, an attempt is made to resolve the fault within the restrictions of the plan or else pass the failure up to the next level. Ultimately, the deliberative system may have to generate a new plan.

One way to achieve this behavior is to pass failures directly to the planner when the lowest level activity manager cannot resolve them. The planner has visibility into all of the goals, subgoals, activities, resources, etc. It can immediately determine whether the failure can be resolved with a fast local patch to the plan or whether it has wide reaching implications that require a major overhaul. This approach is almost certainly more efficient than having managers for each subgoal, since the information and reasoning in the managers effectively

replicates the information and reasoning in the planner, except that the managers are restricted to local reasoning.

Planning and Scheduling

A planner/scheduler generates sequence of coordinated low-level activities (a plan) that achieve a set of high level goals. The plan is guaranteed to satisfy constraints on operability, temporal ordering, resource utilization, etc. that are contained in a declarative plan model. The planner resolves resource contentions and other interactions among activities.

The level of planner autonomy must be scaleable. This allows ground operations personnel to a range of control over the spacecraft, from detailed control for special situations such as post-launch checkout or anomalies that the autonomy system cannot handle, to high-level commanding for nominal operations. The level of autonomy can also be started low and increased as confidence in the autonomy system grows.

The following scale of autonomy is supported by most planning systems. These levels all use the same planner, but vary in the level of control the planner is given over spacecraft operations.

- *A ground-based version of the planner is used to expand goals into an activity sequence (plan). The ground team can then modify the plan as desired (insert or move activities, relax constraints in the model [flight rules] to get highly optimized plan, etc.). The planner will check the plan against its model for constraint violations and inconsistencies, but the ground team is free to override these warnings. The plan is executed by the spacecraft without further expansion by an onboard planner.*
- *Planner generates plan from high-level goals. Ground team can check the plan for accuracy, but otherwise does not modify plan. The plan is uploaded to the spacecraft for execution, and is not expanded further by an onboard planner.*
- *A plan is generated on the ground to some level of detail. An onboard planner expands the bottom-most goals in the plan into detailed activities, and the complete plan is then executed. The ground therefore controls the higher level spacecraft behavior down to some level, and lets the onboard planner worry about the details below that point.*
- *Specify only high-level goals, and let the onboard planner expand them into a detailed plan of activities.*

The architecture should support a planner based either on the ground or on the spacecraft. It should be possible to have both ground-based and onboard versions of the planner within the same mission, and to change the locus of control between these at will (see scale of planner autonomy above). Ground based planning allows more ground control of spacecraft operations. Onboard planning can also allow detailed control by the ground, but when given fuller autonomy, an onboard planner can close loops onboard. This capability is important for events that require the spacecraft to respond quickly to environmental information.

Planning and Execution Cycle

The basic cycle is simple: generate a plan, execute it, and when problems arise generate a new plan that addresses the problems. A problem is any unplanned event that violates the assumptions in the plan, or is about to do so if left unchecked. If a problem has not yet violated the plan, then it can be handled within the reactive execution system. If the plan has already been violated, then the plan must be repaired. The level of replanning (repair) required depends entirely on how globally the problem impacts the plan. If the problem has minor local effects, then a reactive execution system should have enough knowledge to repair the problem without invalidating the rest of the plan. If the problem has larger effects, then resolving the interactions will require a planner. If these effects are still relatively localized, then iterative refinement should be able to repair the plan quickly without impacting unrelated segments of the plan. If the impacts are major, then the refinements will expand to the entire plan, possibly resulting in a totally different plan. Small repairs can be done quickly by most planners. Problems that require a global replan can be computationally expensive regardless of the planner used.

Modularity

The planner model can be broken into modules corresponding to goals, subgoals, activities, resources, etc. This puts all of the knowledge relative to these items in the same place. However, the "modules" will always have tight coupling with each other. By definition, a planner deals with global interactions across subsystems and resources, and these interactions must be captured in the affected modules of the planner model. This leads to tight coupling.

The planning engine has visibility into all of the modules. There is no opaqueness or abstraction barrier. This is necessary to reason about all of the constraints and interactions. The planning problem is essentially a large constraint satisfaction problem (CSP). Just as a CSP cannot be easily divided into independent modules, neither can the planning model. If the CSP could be so divided, it would be easy to solve by decomposition. However, CSPs are generally very difficult to solve and cannot be decomposed. The same applies to planning models, or the knowledge for any deliberative system that needs to reason about the global interactions among the spacecraft subsystems and resources.

Planning for Multiple Spacecraft

The architecture should support goal-based commanding for activities coordinated across several cooperating spacecraft. Among the key issues are:

- *Distributed planning across spacecraft.*

  The idea is to plan various subgoals of the overall planning task on separate processors/spacecraft, with a central processor coordinating the effort. The subgoals all interact, so communication among processors will be necessary (either through the central processor, or directly among the interacting subgoals). [Capability not available yet, but it is being actively researched.]

- *Distribution of failure information*

- *Failure responses may involve several spacecraft.*

- *Coordination of events by other than time-based means (e.g., when you see me do X , then you do Y. But I can't predict exactly when X will happen).*

Coordinating activities across spacecraft requires some level of centralized control and communication of planning and execution information among the spacecraft. The base X2000 architecture should expandable to support multiple spacecraft.

Verification

- Break plan model into modules. Typically, this is everything needed to support a particular goal or sub-goal. Low enough level sub-goals should deal with a single subsystem. Model can be tested from bottom up this way.

- Use scaleable commanding to test from bottom up. Test lowest-level actions first (roughly corresponding to the command dictionary). Then test sub-goals involving interacting subsystems. Keep going to system level behavior by running entire planner.

With sufficiently abstract simulator, can test from top down. Provide high-level goals and execute resulting plans to make sure system behavior is working. Can be done before detailed sub-system s/w, h/w and sims have been delivered.

Model validation

The autonomy engine will guarantee that the spacecraft behavior is correct with respect to the declarative models, assuming that the engine is bug-free (this is true for all autonomy systems, not just the planner). So one validation task is validating the engine.

The second validation task is validating the knowledge in the models themselves.

6.1.2 Versatile Task Specification

Sequential

Timed Execution

Event Chaining

Concurrent

Coordinated

Competing

Priority Execution

Combination

6.1.3 Level Of Autonomy

[See “Autonomy” in Appendix A — Definitions]

Spacecraft need to be commandable over a wide range of levels of autonomy, ranging from traditional open-loop time-based sequences to fully autonomous responses to high-level goals. Here's a strawman classification of different levels of autonomy.

> Level 0: Traditional open-loop time-based sequences, e.g. "Send DEVICE_X_POWER_ON_COMMAND message at time T1."
>
> Level 1: Closed-loop commands that achieve a condition at a point in time, e.g. "Turn device X on (and verify that it is in fact on)."
>
> Level 2: Closed-loop commands that maintain a condition over a time interval, e.g. "Turn device X on at time T1 and make sure it stays on until time T2."
>
> Level 3: Closed-loop maintenance of conditions over temporal intervals whose endpoints are specified with respect to runtime events, e.g. "Turn device X on no more than five seconds after event E."
>
> Level 4: Closed-loop maintenance of multiple conditions over time intervals, with a mechanism for detecting and dealing with conflicting conditions, e.g. "Turn device X on after event E1, and turn it off after event E2. Make sure it stays on at least 10 seconds," in the case where E1 and E2 are less than ten seconds apart.
>
> Level 5: Closed-loop maintenance of multiple conditions at higher levels of abstraction, e.g. "Configure for attitude knowledge acquisition with hot backup."
>
> Level 6: Automatic execution of complex networks of temporally constrained conditions, e.g. "Perform autonav imaging," or "Do orbit insertion."
>
> Level 7: Very high-level goal-based commanding with autonomous prioritization, e.g. "Do a site survey and send back the most interesting data," or "Fulfill as many of the following observation requests as possible."

Note that this is a taxonomy of *commands* sent to a spacecraft, not of autonomy technologies.

Resource Management

Planning and Scheduling

6.1.4 Execution Logging

6.2 Hardware Management

6.2.1 State Tracking

6.2.2 Configuration Control

6.2.3 Consumable tracking

6.3 Data Management and Telemetry

[This is a start, but more is needed to complete the picture.]

An unmanned spacecraft is not a desktop, not a workstation, but a robot that responds to mission events. That robot is connected through a communications network to monitor and control systems; the network happens to be one for which at least one of the links requires wireless transmission through space, and the aggregation of the robot's own software and the monitor and control systems on earth is the UFGA.

In order to truly unify the flight and ground systems, and thereby minimize the impact of flight/ground design trades and the cost of migrating functionality between the spacecraft and the ground, it is valuable to use the same mechanism for communication between flight software and ground software as is used among the flight processes. If the flight interprocess communication mechanism is asynchronous message passing, then ideally that same mechanism should be used for communication between flight software and ground systems.

Implementations of layered deep space communication protocols, including a reliable deep space transport layer, will support this operational model. Reliable transport entails efficient transmission acknowledgment and data retransmission on partial loss or corruption. Delegating responsibility for this reliability to standard, reusable protocol implementations will reduce mission cost and risk and will simplify flight software, insulating it from any functional differences between on-board and deep-space interprocess communication. Operational differences will of course remain — time-critical closed control loops clearly can't be deployed across the space link — but modern deep-space communication protocols can help reduce the problem of ground/space software migration from one of software compatibility to one of system configuration.

It's important to note that the client/server model of interprocess communication is less amenable to this architecture. Issuing true remote procedure calls over a space link is

impractical due to the very great distances separating the communicating procedures: the sender of an RPC would spend far less time executing than waiting for propagation (at the speed of light) of the procedure invocation and the receiver's response.

Asynchronously passed messages, on the other hand, are well suited to the delivery of continuous, open-ended streams of time-tagged engineering data values and instrument observations. In this model of operations, queues of messages serve many of the purposes for which files have been used in the past (both on the spacecraft and on the ground). However, in order to serve those purposes, those message queues must be no less robust than files; in particular, they must not be destroyed when power is withdrawn from dynamic memory. That is, the message queues used for flight/ground communications need to reside in persistent storage media. The random data access made possible by modern on-board storage technology (such as solid- state recorders) provides the flexibility needed to implement complex persistent data structures such as message queues.

None of this addresses the fundamental problem of limited space link bandwidth. As the requirements for meaningful science data return increase it becomes increasingly necessary for the spacecraft to convey better information without sending more. Among the strategies for accomplishing this goal are downlink management, data compression, and data summarization.

Downlink management doesn't reduce data volume but instead just maximizes transmission efficiency. All data to be downlinked are categorized, and associated with each category are a priority and a bandwidth allocation percentage. High-priority data (typically relating to spacecraft health) are downlinked before all lower-priority data. Among messages of the same priority, access to the space link is apportioned according to bandwidth allocation percentage; this encourages the sources of those messages to issue them in order by descending usefulness. The importance and usefulness of the data transmitted over any given interval are thereby increased even though the actual data transmission volume is not.

Individual data items can be compressed in either "lossless" or "lossy" fashion as described elsewhere in this document. The reduction in bandwidth consumption resulting from data compression, and the corresponding increase in efficiency of space link utilization, can be dramatic.

Finally, a potentially even more powerful way to increase link efficiency is simply to send the products of data analysis and summarization rather than the raw data itself. On-board data summarization algorithms are still a research topic, and they rely on the availability of large on-board data storage resources and spare processing capacity, but as flight missions range further from Earth and available transmission bandwidth diminishes the costs of enabling this in-situ analysis are increasingly justified.

6.3.1 File Management

Uplink

Downlink

6.3.2 Telemetry

Scheduled

Event Driven

6.3.3 Data Management

Compression

Culling

Data Mining

6.3.4 Mechanism for Feedback into Subsequent Activity Plans

6.3.5 “Beacon mode”

6.4 Guidance, Navigation, and Control

6.4.1 Pointing System

6.4.2 Navigation

6.4.3 Maneuver Planning

6.4.4 Dealing with Constraints

6.5 Power and Thermal Management

6.6 Telecom

6.7 Science

6.8 User Interface

6.9 Test

o

7 SOFTWARE VERIFICATION

[This section needs to be coordinated with other related sections of the document.

Also, there were two contributions to this section (both below) which need to be reconciled and then merged.]

7.1 Cut 1

Verification is the process of checking that the software implementation satisfies the requirements. Verification is much more than just after-the-fact unit and system testing. It begins with requirements and extends beyond final software delivery to in-flight behavior auditing. Verification takes several forms and appears in different places and different phases of the development process.

This section addresses a few key issues for verification. Some of the issues listed below have no direct architectural impact, but are included to encourage awareness of verification impacts on the design, development, and testing processes.

- Testable requirements
- Scenario specifications
- Detailed simulation environment
- Unambiguous interface definitions
- Embedded constraint tests
- System-level behavior auditing
- Safety kernel
- Incremental builds and automated regression testing
- Code inspections

7.1.1 Testable requirements

As far as possible, each requirement should be stated clearly enough that it can be verified, preferably without human involvement. Requirements that are too abstract or vague to be

verified should be decomposed into specific testable sub-requirements. Requirements that cannot be tested without a human in the loop should raise a red flag because that complicates automated regression testing and suggests a vague or excessively abstract requirement.

7.1.2 Scenario specifications

In order to construct good system-level tests, system engineers should define not only the nominal scenario for a mission but also the type and timing of failures that will particularly stress the software's recovery mechanisms. System engineers should make the following things explicit: boundary conditions, outlying but acceptable variations in values and event times, and type and timing of particularly likely failures.

7.1.3 Detailed simulation environment

The flight/ground simulation environment should be detailed enough and accurate enough to "fly" the mission, as well as to verify individual subsystems. Such an environment keeps designers and developers honest and does not allow them to ignore details that will harm them late in the integration phase when the real hardware appears.

Simulation models should be sufficiently accurate that spacecraft engineers respect the results of testing with simulated subsystems. Such models should: (a) interact appropriately with other simulation models through electrical/mechanical/thermal/etc. pathways, (b) support reasonable failure modes, (c) report when a subsystem is being mishandled by the flight software, (d) support checkpointing, and (e) support time-warping.

7.1.4 Unambiguous interface definitions

A component interface should be defined precisely enough that it can be checked statically (as in a compiler check of the number and type of arguments) and dynamically (as in range checks on valid values and as checks on protocol adherence). Note that dynamic checks that are active during ground testing may (optionally) remain active during the mission.

7.1.5 Embedded constraint tests

To simplify debugging, errors should be detected as close to the source as possible. This means that every component should include constraint tests on its inputs as well as self-tests on its own computations. When a constraint is violated during execution, this helps enormously in localizing the site of the error. To support such tests, the run-time environment must provide a standard mechanism for reporting constraint violations and specifying a reaction (e.g., abort, warn, or log).

7.1.6 System-level behavior auditing

In addition to having a mechanism for reporting locally testable constraint violations, the run-time environment must also provide a standard mechanism for making selected activities visible to a system-level behavior auditor. Typically, the selected activities will

include events, state changes, measurements that are relevant to the verification of flight rules (e.g., "never point camera at sun") as well as checking of design artifacts (e.g., "initiate-turn command should be followed by turn-complete confirmation within 2 minutes"). In addition, retries and recovery actions should be reported so that it is known that they are happening.

7.1.7 Safety kernel

During a mission there are some errors for which detection alone is not enough; some actions must be suppressed and some inactions must be corrected in order to prevent damage to or loss of the spacecraft. Accordingly, there should be a "safety kernel" that sits logically between the hardware and other flight software. Normally this safety kernel is transparent to and non-interfering with the higher-level software, but in cases where a spacecraft hazard is imminent, it may suppress actuator commands or actively command the spacecraft into a safer state. This safety kernel is an important safeguard against mistakes in flight software as well as mistakes in uplinked commands.

7.1.8 Incremental builds and automated regression testing

As much as possible, functionality should be added to a system incrementally, as a sequence of correctness-preserving transformations. For example, the system should be rebuilt every night and tested against an ever-increasing suite of regression tests. This approach sometimes requires careful planning and coordination among members of the development team, but it is usually worth it because it avoids the extremely difficult problems of integrating and testing multiple new/revised components.

7.1.9 Code inspections

Code inspections can be valuable in finding errors and omissions early in the development process, but they are also time-consuming if conducted as formal meetings of several people. A less intrusive approach to code inspections is to give the code to one or two colleagues for inspection, with comments expected within a week or two. Besides finding errors, inspections tend to encourage a consistent style of coding and commenting, and reduces project vulnerability to the loss of a programmer.

7.2 Cut 2

7.2.1 Verification and Validation

The current conventional wisdom is that the software developed for future missions will be so complex that traditional testing and validation methods will no longer apply, and radically new approaches will be needed. While it is true that testing methodology will have to change, this change does not have to be a radical departure from traditional methods. In fact, most of the features that make testing difficult already exist in "traditional" spacecraft control software. For example, it is commonly believed that autonomy software is non-deterministic, and that this presents an extraordinary new testing challenge. In fact, all autonomy software developed at JPL to date is completely deterministic. It is true that the

system behavior is not always predictable, but this unpredictability arises because of the unpredictability of the environment, not a lack of determinism in the software. This situation entirely identical with that in traditional attitude control software, whose detailed behavior is not predictable a priori, but which is nonetheless deterministic and testable.

In fact there is no fundamental difference in testability between autonomy software and attitude control software. Both are closed-loop control mechanisms designed to maintain certain system invariants in the face of external disturbances. In the case of ACS, the system invariant is spacecraft attitude, and the disturbances are external torques and imprecision in the attitude control actuators and sensors. In the case of "autonomy" systems, the system invariants are constraints on the system state expressed at higher levels of abstraction, and the disturbances are hardware failures.

The key to verifying both ACS and autonomy systems, and indeed any complex software system, is to enumerate the design invariants the system is intended to maintain, and the circumstances under which they are to be maintained, and then verify that the system does indeed maintain those invariants. This verification can be done empirically through empirical testing (which can be exhaustive if the term is taken to mean insuring coverage over the range of invariants and disturbances, not control branch points) or formal methods.

The key to this approach is enumerating the invariants and disturbances against which one wishes to verify the system's behavior, which can be quite complicated.

One helpful design principle which can make this job easier is Law of Cognizant Failure [Gat91]. The LoCF states that, instead of designing system that never fail, one should instead design systems that detect failures when they occur. In other words, all system invariants should be of the form, "The system is guaranteed to either achieve X, or signal that X has not been achieved." Cognizant failure is useful because it is much simpler to design systems that are guaranteed to detect failures than systems that are guaranteed to avoid them. If all failures are cognizant, then the system can be designed to automatically recover from all failures through layered recovery procedures. At the top-level of the recovery hierarchy is a more-or-less traditional safe mode, which the spacecraft enters when all other avenues of recovery have been tried and failed. This recovery of last resort is exhaustively tested using traditional methods. This approach can guard against both hardware and software failures, and provide confidence in the overall reliability of the system even if it contains untested components.

o

8 Hardware Requirements

8.1 Modelable Behavior

8.1.1 “Delta” Commands Restrictions

8.1.2 Time

Synchronization

Time Tagging

Commands

Data

8.2 Self Safing

8.2.1 Reset to benign, passive state

8.2.2 Regular software access necessary to sustain active states

8.2.3 Protected access to critical functions

8.3 Fault Protection

8.3.1 Internal detections and responses

Built In Tests shouldn’t lie

No ambiguous status

E.g., Rollover

No unobservable critical faults

8.3.2 Containment regions

8.3.3 Isolation

Independent access to isolation mechanism

8.4 Redundancy

8.4.1 Symmetry

8.4.2 Independence

8.4.3 Cross Strapping

8.5 Bus and Network Issues

8.5.1 Master Selection

8.5.2 Masquerading Terminals

o

9 Appendix A — Definitions

9.1 Software Architecture

"The primary objective of architectural design is to develop a modular program structure and represent the control relationships between modules. In addition, architectural design melds program structure and data structure, defining interfaces that enable data to flow throughout the program."

[Roger S. Pressman, "Software Engineering: A Practitioner's Approach", Third Edition, 1992.]

"Architectural design involves identifying the software components, decoupling and decomposing them into processing modules and conceptual data structures, and specifying the interconnections among components."

[Richard E. Fairly, “Software Engineering Concepts”, 1985]

9.2 Autonomy

Colloquially, a system is "autonomous" to the extent that it accomplishes tasks that previously required a human-in-the-loop. For example, a flight/ground system that automatically diagnoses spacecraft faults is more autonomous in that respect than one that doesn't. There is no absolute autonomy scale, just relative differences within functional areas.

The scope of potential new autonomous capabilities is as broad as the range of tasks currently performed by people. Opportunities exist in sequence planning, navigation, fault protection, engineering data summarization, science data processing, and software verification.

Capabilities that make a system more autonomous can exist in ground software as well as flight software. Some capabilities can be situated in either place, such as high-level activity planning; others must necessarily be onboard due to reaction-time constraints (stuck thruster) or limitations of telecommunication data rates (feature recognition in numerous images).

Architecturally, autonomous "agents" replace human-in-the-loop calculation/reasoning and must therefore support the necessary interfaces and close the loops. Given the many different kinds of knowledge and reasoning that humans bring to problem solving, there is no standard architecture for agents, but two basic design principles are widely used. First, knowledge about the problem domain is represented in a declarative form. This inspectable knowledge base describes *what* is known, possibly in the form of models or rules or cases. Second, the method for *when* and *how* to apply the knowledge is defined in a [deterministic] inference procedure. This separation of knowledge and inference procedure yields a readily inspectable base of knowledge whose clear semantics stem from a formally defined inference procedure

9.3 Object-Oriented Software

An *object* is a collection of data items and subroutines that operate on those items. The data items are known as the object's *slots* or *instance variables*, and the subroutines are known as the object's *methods*.

An *active object* is an object with its own thread of control, i.e. an object with a program counter and a stack or continuation chain included among its instance variables. Active objects are sometimes called *processes* (if the object shares no state with any other object), *threads* (if the object shares global state with other threads), or *tasks* (a term used in vxWorks, where it means the same thing as thread).

An object *class* is a description of a set of objects which share common methods and whose instance variables share a common structure. An object which is a member of the set described by a class is known as an *instance* of the class. Some

programming languages allow the properties of one class to be defined in terms of the properties of another, a feature known as *inheritance*. A complex collection of classes defined in terms of one another is known as a *class hierarchy*. In some programming languages, classes are themselves objects, and thus can be instances of a *meta-class*, i.e. a class whose instances are other classes.

A *first-class* object is an object that can be manipulated as a monolithic entity, i.e. the entire object can be passed as a single argument to or returned as a value from a method or subroutine. A *distributed object* is an object whose methods run on multiple processors, and/or whose slots reside in multiple memory systems.

A program is *object-oriented* to the extent that any persistent datum in the program is a member of an object, and that datum is accessed and manipulated exclusively through that object's methods. In other words, a program is object-oriented when the set of functions that can operate on any particular data item is explicitly or implicitly enumerated. Object-orientedness is a continuum, not a discrete property.

Note that it is not necessary to use a so-called "object-oriented" programming language to write object-oriented programs. Objects, both active and passive, can be constructed in any programming language, though it may require more effort on the part of the programmer in some languages than others.

Caveat: It is important to distinguish the general concept of object-oriented design from any particular instance of an object-oriented design, and to keep in mind that using object oriented design methods is by itself no guarantee of producing a good design.

o

10 Appendix B — Examples

10.1 Model Based Software Design

There are two examples that serve to illustrate the idea of model-based software design.

10.1.1 Fault Protection Monitoring

In DS1 program the model for a fault protection monitor is a specification of the data transformation and filtering to be applied on raw sensor data for the purpose of detecting a specific fault symptom. The model assumes an architecture where data transformations are domain-specific mathematical functions (e.g., the Cassini-style phase portrait rotation of control error and rate of control error), while data filtering operations are either custom-made or reused from a library of generic data filtering components (e.g., threshold checking and detection, transaction success/failure tracking, nominal range tracking).

Fault protection on DS1 relies on one code-generation tool to transform each monitor model into a suite of products ranging from flight software source code, telemetry packet definitions, software interface headers, and script-based unit test driver software. The

specific aspects of the target software architecture are not embedded inside the code generator but are instead kept into separate models called templates that describe the syntax and format of each specific product to be generated such as source code. In this approach, reusability can be exercised in multiple ways such as: writing new models for building new monitors, writing new template files to produce different products from each model, adding new data-filtering components to extend the range of symptom detection techniques, and adding new modeling constructs to incorporate additional information about purpose and functionality into the model and to express how this additional information is used in code generation.

10.1.2 Software State Charting

State charts is a well-known organizing paradigm to describe behavioral information. A state chart can be seen as being a piece of the model of a software component. In that context, a state chart software code generator is a model-based tool that can be exploited towards promoting reuse of information to reduce redundant expressions of the same model in a variety of forms such as specifications, software implementation, functional documentation, and test drivers.

In DS1, statecharting is used to describe the design of data filtering components for symptom detection in fault protection monitoring. A code generator tool then produces flight code for each data filtering statechart. This achieves the reuse of information to avoid unnecessary duplication of effort in terms of design and implementation: the same statechart fits both purposes.

Furthermore, the same code generator also produces Java code which is then used along with a simple Java GUI to make a standalone unit test simulation of the data filtering statechart. In this manner, the same statechart is reused at the process level not only for software development purposes but also for software documentation, test and training purposes.

o

11 NEW MATERIAL NOT YET INCORPORATED

11.1 Modeling

This is a checklist for hardware modeling. It describes the kinds of information captured in design models that subsystem vendors must prepare.

11.1.1 Introduction (For Vendors)

An autonomous spacecraft has goals. It makes plans to accomplish those goals, it commands the subsystems to carry out those plans, it monitors the subsystems to confirm command execution, and when misbehavior is detected it isolates the fault to one (or a few) candidates and then performs appropriate recovery actions.

The autonomy software that does all of this depends on using knowledge of each subsystem — knowledge such as its modes of operation, commands, operational constraints, observable measurements/sensors, kinds of faults, and recovery actions. The checklist below enumerates the kinds of knowledge that we need from you. Notice that the level of detail that we want is roughly equivalent to high-level design specifications with emphasis on the information that flows into and out of the subsystem; a 1-page block diagram is often the right level of detail for looking inside the subsystem.

11.1.2 General

- Please define all acronyms (for the acronym-challenged among us).

11.1.3 Architecture, Inputs, Outputs, Design:

- Provide a block diagram where the subsystem is a black box showing:
  - Physical information pathways (VME, 1553, 1773, analog, digital, serial, optical, radio) and where they connect to the rest of the spacecraft;
  - Physical relationship of this subsystem w.r.t. the spacecraft frame and other major subsystems.

- Define all inputs to subsystem:
  - Software commands via each bus (VME, 1553, 1773, …);
  - Associated command parameters;
  - Other signals via non-bus pathways (analog, serial, digital).

- Define all outputs from subsystem:
  - Status and data via bus and non-bus pathways;
  - Electrical signals via non-bus lines.

- Describe the design of the subsystem itself through:
  - A block diagram showing its sub-subsystems and interconnections for power, communications, and sensors;
  - A brief description of subsystem BEHAVIOR (for commanding, monitoring, and inferring its state);
  - A brief description of subsystem PURPOSE (for modeling its function, evaluating degraded capability, and planning its usage accordingly).

11.1.4 Sensors / Observables:

- What is the rationale for having and placing each sensor?

- Describe the status and data variables that can be read.

- For each sensor/variable:
  - How is it read?
  - What kind of value does it return: bit, integer, float, state?
  - What are its absolute minimum and maximum values?
  - What is its nominal range, and does it depend on operating mode or other variables like temperature, pressure, voltage, etc.?
  - How noisy is the returned value?
  - What kind of noise filtering or smoothing should be applied?
  - How accurate is the sensor at launch? a month later? a year later?
  - Is there an independent way to corroborate its value?
  - Does it degrade in a predictable way?
  - What's the probability of failure?
  - When it fails, how is the measured value affected: is it stuck high, stuck low, zero, erratic, or unchanging?
  - If it returns a value outside the nominal range, is there a simple way to distinguish between sensor fault versus system fault?

11.1.5 Monitoring:

- Is there a direct (or indirect) way to confirm that each command is being (or has been) carried out?

- Is there a way to determine the current operating mode?

- When an autonomic (non-commanded) transition occurs, how is that reported?

- Are there clear boundaries of normal operation?

- Are there clear signatures of abnormal operation? For example, is there a combination of sensor readings, possibly observed over time, that signifies a fault?

11.1.6 Modes & Transitions:

- Draw a state-transition diagram where:
  - Each node represents a high-level operating mode of the subsystem including startup, shutdown, normal, degraded, and restart modes, as appropriate;
  - Each link represents a legal transition from one mode to another.
  - What is the initial mode following power-on-reset?
  - Which transitions are commanded and which are autonomic?
  - Can the subsystem be commanded into a "self-test" or "diagnostic" mode where the subsystem goes off-line and checks itself for faults?
  - Are there modes that should NOT be used?
  - If the behavior within a high-level operating mode can best be described with another state-transition diagram, please provide that.

11.1.7 Operational Constraints:

- What constraints exist across all modes (e.g., sun exposure, thermal, power, etc.)?
- For each command, are there situations when it should NOT be issued?
- For each mode, are there commands that should NOT be issued?
- Are there SEQUENCES of commands that should NOT be issued?
- Should certain commands be avoided?
- Should the time in certain modes be minimized?
- For each command, are there any timing constraints?
- Are there preconditions that should be met before certain commands are issued?
- Are there startup and/or shutdown delays for some components?
- Are there components whose respective commands (or states) must be coordinated?
- Are there time constraints on any modes or transitions?

11.1.8 Resource Usage, Environmental Impact, Life Span:

- Quantify nominal resource usage (power, propellant, bus occupancy, SSR memory, etc.), e.g., "when on, consumes 50W on the 28V bus".

- Describe environmental impact on other subsystems (e.g., dissipates 100W heat, causes vibration, radiates electromagnetic fields, causes voltage spikes on power bus, etc.).

- Quantify life span of components in term of allowed operations and environmental exposures.

- Does execution of certain commands or duration within certain modes:
  - Consume significant resources (e.g. power, propellant)?
  - Interfere with other subsystems?
  - Cause physical degradation?

11.1.9 Faults, Failures, Recoveries:

Definitions: A "fault" is a defect in a component of the subsystem; a "failure" is an observable manifestation of a fault.

- Fault/failure/recovery information can be described in a table whose column headings are: current mode, observed failure, possible fault, [relative] probability, recovery action, and next mode.

- For each fault:
  - Is there a way to confirm it, possibly by examining several measurements or observing behavior over time?
  - How urgent is it to perform recovery?
  - Can it cause consequential faults?

- For each recovery action, what are the undesired consequences, i.e., the effects other than fixing or bypassing the fault?

- Are there plausible double-faults that should be considered?

- Does the subsystem ever initiate AUTOMATIC fault recovery? If so, how can it be determined that that is happening?

11.1.10 External / Exogenous Events:

- What are the exogenous events that affect this subsystem? Include external stimuli, time-outs, and faults.

11.1.11 Complexity / Cost Estimates:

It could be useful to provide estimates of complexity or cost for a particular sensor or functionality. There may be opportunities for subsystem designers to reduce costs by taking advantage of the relatively sophisticated system-wide monitoring, diagnosis, and recovery software provided by full autonomy. For example, if a subsystem contains a costly sensor for measuring a value that the software can infer by other means, then system engineers may elect to eliminate the sensor.

11.2 Architecture

11.2.1 Introduction

11.2.2 Document Objectives

11.2.3 Design Guidelines

11.2.4 An Approach to Layered Design

11.2.5 The Software Design

Summarized Architecture Guidelines

In order to ensure that the X2000 architecture covers all of the architecture guidelines, the guidelines are enumerated below. Also included is the chapter, section and subsection number of the previous chapters from which the guideline was surmised:

[Enumerated list of s/w design guidelines]

In the following sections we prescribe a common basis for the X2000 architecture so as to satisfy the architecture guidelines. Throughout we specify which of the design guidelines the prescribed architectural component impacts using the above enumerated list with the notion '{<guideline>}'. In this way it will be clear that the guidelines are thoroughly addressed by the architecture. That is not a guarantee that the subsequent spacecraft software design and implementation will embody the design guidelines. Although, without a suitable architecture, the spacecraft software could at best only coincidentally satisfy the guidelines.

The Object-Oriented Approach

The object-oriented approach has emerged as the dominant approach to the software engineering of large, complex, mission-critical software applications. [...]

The 'object' in object-oriented is the software model for a physical or conceptual 'external object'. The software object is often confused with the external object that is models. This is desirous because it is an indication that the software model has 'covered' the external object accurately. Any disparity between the two objects warrants concern.

Like external objects, objects have identity, state and behavior. [...]

A spacecraft is a physical object that is amenable to modeling as a software object. Spacecraft have identity like 'the Cassini spacecraft'. Spacecraft have state like mass, cost, subsystems and life expectancy. Spacecraft have behaviors to turn, to navigate, to return

science data, and to shutdown. Spacecraft are certainly large (if not physically, programatically), complex and mission critical.

A spacecraft is the pinnacle in a abstraction hierarchy that extends down to the bits in a memory location, the amp-hours in a battery, the Kelvins in a catbed heater, and the grams in a fuel tank.

Conceptual Framework

Basic concepts provide a basis for communication on software architecture issues. In what follows we quote definitions [Booch94] for several concepts that are generally important in software engineering and that are often the focus in an object-oriented design process. Descriptions can be found in various references; descriptions on their applicability to spacecraft is current lacking, herein.

Abstraction: "An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer."

Encapsulation: "Encapsulation is the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation"

Modularity: "Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules."

Hierarchy: "Hierarchy is a ranking or ordering of abstractions"

Typing: "Typing is the enforcement of the class of an object, such that objects of different types may not be interchanged, or at the most, they may be interchanged only in very restricted ways."

Concurrency: "Concurrency is the property that distinguishes an active object from one that is not active"

Persistence: "Persistence is the property of an object through which its existence transcends time (i.e. the object continues to exist after its creator ceases to exist) and/or space (i.e. the object's location moves from the address space in which it was created)."

Modeling

Models in an object-oriented approach are derived from the vocabulary of the domain for which the software is employed. This is after all one of the significant advantages of object-orientation in that it provides a common vocabulary for software developers, domain experts and users. Given a common vocabulary miscommunication is less likely; one impact is likely a reduction of errors in the software product.

Using model-based design for spacecraft software will, for the first time, provide an accessible, uniform vocabulary for discourse between disparate divisions. The vocabulary is grounded upon the entities and activities in spacecraft and of space, exploration and science. The models thus reflect the core capability of JPL.

High-level models are inherently free from obsolescence because they are based on the physical and conceptual world, that is the domain. Thus, if it is appropriate to model a spacecraft as having attitude control, navigation and camera subsystems today it is highly likely that, short of a technological revolution, spacecraft will still have the same subsystems twenty-five years from now. This is not to say that the specifics (read, implementation) of navigation, for example, will be unchanged during that interval. It is expected that implementations change but the interface, captured by the model, likely will not.

Lower-level model will naturally reach obsolescence. In particular, hardware specific device models invariably change as technology progresses and even more frequently as versions change. An example might be new imaging technology leading to more accurate star trackers. However, the model just above the hardware specific model, say the generic 'star tracker' model, likely need not change because, say, the fundamentals of tracking stars are fundamental.

Given good models, the stability of the domain is reflected in the stability of the software models.

Modeling Languages

Software models must be expressed in a software modeling language. To be useful the modeling language must be: 1) expressive enough to capture the domain, 2) precise enough to reduce ambiguity in expression, 3) process independent, and 4) understood by all team members. These characteristics enable a modeling language as an effective tool.

The "Unified Modeling Language" (UML) is a consolidation of several modeling languages. Each of the modeling languages: Booch (Booch), OMT (Rumbaugh), and OOSE (Jacobson) had certain strengths and weaknesses and had been adopted by large numbers of developers in the object-oriented software community. UML builds on their successes.

From the UML v1.0 Summary document - the unification effort established four goals: "1) to model systems (and not just software) using object-oriented systems, 2) to establish an explicit coupling to conceptual as well as executable artifacts, 3) to address the issues of scale inherent in complex, mission-critical systems, and 4) to create a modeling language usable by both humans and machines." The UML does not address issues in the software development process; although the UML authors promote a process that is "use-case, architecture centric and iterative and incremental."

The UML modeling language ought to be adopted by X2000 as the standard modeling language.

Having a standard modeling language does not prevent use of specialized languages. Standardization simply provides a basis for communication Yet, within different teams or technologies, other modeling languages might be required. Two examples are the model languages used by the DS1 PS and the DS1 MIR teams. Still, ideally one language fits all although not without certain hardship. Multiple languages reduce readability and increases specialization at the expense of shared experience.

Amongst different tools, choosing one that is standardized and proven can reduce risks.

Architecture

Software architectures consist of both logical and physical views [Booch94]. The logical view of a system "serves to describe the existence and meaning of the key abstraction and mechanisms that form the problem space" while the physical view of a system "describes the concrete software and hardware composition of the system's context or implementation. Both views have both static and dynamic components.

In the object-oriented domain the logical view is comprised of the classes, the class relationships, the objects, and the mechanisms of collaboration between the objects. The physical view is a specification of where the classes and objects are declared, what processors and devices exists, how are the processes allocated between processors and what scheduling mechanisms are employed.

The physical processors and devices do not as a rule appear in the logical view. For example, in a user interface with a class such as <window>, the processor upon which the <window> is displayed would not be modeled explicitly (provided active and passive objects existed explicitly - see the definitions below). However, in a robotics application, like spacecraft, where resources are constrained and interaction with the environment is paramount it is perhaps a necessity to model the processor itself (say for mass, thermal, and memory properties). In these cases, caution must be observed to maintain the distinction between logical and physical views.

Relation to 'Ground'

Although it is rather premature within this document to discuss 'ground' and 'ground software', there is one important point to be made in light of the architectural divisions of physical and logical views. In a physical view a 'spacecraft' consists of one or more processors on-board and one or more processors off-board, on the ground. The physical view will have two large boxes one labeled 'flight' and the other labeled 'ground.' In the logical view flight software and ground software are both spacecraft software and what functionality is where is thus largely irrelevant. The logical view will not have any box labeled 'ground' but instead might have boxes labeled 'debugger' or 'database' or 'display.' The distinction between physical and logical views is an empowering one and leads naturally to the unification of flight and ground software.

Definitions

The following definitions are from the 'Unified Modeling Language' or from Booch94 ("Object-Oriented Analysis and Design with Applications"):

General

[... specific relevance ...]

OBJECT: An entity with a well-defined boundary and identity that encapsulates state and behavior. State is represented by attributes and relationships, behavior is represented by operations and methods. An object is an instance of a class.

CLASS: A description of a set of objects that share the same attributes, operations, methods, relationships, and semantics. A class is an implementation of a type.

METHOD: The implementation of an operation ('a service to effect behavior'). The algorithm or procedure that effects the results of an operation.

GENERIC/VIRTUAL FUNCTION: An operation upon an object. [...] implemented through a set of methods declared in various classes related via their inheritance hierarchy

MESSAGE: An operation that one object performs upon another. The terms 'message, ' 'method,' and 'operation' are usually interchangeable [In what follows, we avoid 'message' in favor of 'method']

EXCEPTION: An indication that some invariant has not or cannot be satisfied.

CLIENT: An object that uses the services of another object, either by operating upon it or by referencing its state.

SERVER: An object that never operates upon other objects, but is only operated upon by other objects; an object that provides certain services.

Concurrency

[... specific relevance ...]

THREAD-OF-CONTROL: A single process. The start of a thread of control is the root from which independent dynamic action within a system occurs; a given system may have many simultaneous threads of control, some of which may dynamically come into existence and then cease to exist. Systems executing across multiple CPUs allow for truly concurrent threads of control, whereas systems running on a single CPU can only achieve the illusion of concurrent threads of control.

ACTIVE OBJECT: An object that encompasses its own thread of control.

PASSIVE OBJECT: An object that does not encompass its own thread of control.

SEQUENTIAL OBJECT: A passive object whose semantics are guaranteed only in the presence of a single thread of control.

BLOCKING OBJECT: A passive object whose semantics are guaranteed in the presence of multiple threads of control. Invoking an operation of a blocking object blocks the client for the duration of the operation.

CONCURRENT OBJECT: An active object whose semantics are guaranteed in the presence of multiple threads of control.

SYNCHRONIZATION

[... specific relevance ...]

SYNCHRONIZATION: The concurrency semantics of an operation. An operation may be:

SIMPLE: only one thread of control is involved.

SYNCHRONOUS: An operation commences only when the sender has initiated the action and the receiver is ready to accept the method. The sender and receiver will wait indefinitely until both parties are ready to proceed.

TIMEOUT: The same as synchronous except that the sender will only wait for a specified amount of time for the receiver to be ready.

BALKING: The same as synchronous except that the sender will abandon the operation if the receiver is not immediately ready. The same as timeout with a time of zero.

ASYNCHRONOUS: A sender may initiate an action regardless of whether the receiver is expecting the method.

Cautions

Within JPL is often observed that the holy grail of flight software architecture is the specification of a message passing paradigm for a uniform communication interface between all software modules (including ground as well). This observation belies a misunderstanding of the role of message passing in a software architecture and more importantly the different communication requirements in the hierarchy of flight software functionality.

1) Message passing is a specific implementation of inter-object communication. There is never a need to elevate any one implementation to the level of interface as that unnecessarily constrains the interface.

2) Closing control-loops occurs at many temporal levels with each level imposing a different performance requirement on the inter-object communication (memory use, speed, reliability, etc.

Specifically, in terms of the prior definitions, a message queue is one of several implementations for a asynchronous method between two active objects. It is nothing more than that which leaves elevation of message passing to the interface as an undo constraint of the inter-object communication mechanism.

Object-oriented development is more than just knowing C++, Java or CLOS. It is a discipline in itself for which not many people are adequately trained. In spite of the large software engineering component within the JPL workforce is probably safe to say that JPL faces a critical shortage of properly trained software engineers. Given that flight projects have traditionally been hardware-centric, the shortage is particularly acute.

A Spacecraft is a complex beast and it is not likely to fit neatly into one paradigm. Different views, approaches and implementations are required under different situations and times. It is inappropriate to place arbitrary restrictions on a design because of some desire to fit into a particular paradigm. The best that can be hoped for is that ones tools are general and expressive enough to capture the domain and that ones design expresses enough encapsulation to allow 'arbitrary' implementation.

Design Space

The X2000 s/w architecture's logical view is conceptually a multi-dimensional 'design space.' Each axis in the design space represents a largely independent aspect of the s/w

design. The coordinate along any single axis is specified appropriately for that particular axis. Because the design space is for the architecture's logical view, the space deals solely with software models. Thus physical view concepts, such as of processors or devices, are not represented in the design space.

The independence allows One or two aspects often

The axis are: functionality, faults, hardware, simulation and mission; each is discussed below

Functionality

The primary axis in the X2000 design space is the 'functionality' axis. This axis models the hierarchy of software functionality without consideration of hardware, simulation, mission requirements or even faults. As such, the axis models the ideal, generic spacecraft upon which spacecraft systems engineers, mission planners and science working groups base their analysis upon. Thus, an implementation of this axis alone should be useful as a tool for these three groups.

In spite of its simplicity, the functionality axis captures much of that is important in spacecraft software.

Along this axis are four coordinates each of which is described subsequently.

Drivers: The origin on the functionality axis is comprised of 'drivers' which are the software models for generic hardware devices. Examples of drivers include: valve, sensor, switch, battery, heater, waveguide, antenna, gimbal, bus, gyroscope, solar panel, lens, camera and engine. This drivers coordinate is itself also hierarchical: engines might be composed of valves, nozzles, fuel tanks and propellant; cameras might be composed of lenses, imagers and film; solar panels might be composed of wires, ribs, current sensors, actuators and amorphous silicon chip arrays. As models of generic devices, drivers will not generally be detailed to the level of 'bits in a control word' or 'remote terminal number on a 1553 bus' or 'memory-mapped, off-board VME memory.' Such details are implementation, not interface.

Drivers need not correspond to a physical device even though drivers, as defined above, model generic hardware. That is, drivers can correspond to virtual hardware thereby allowing the hardware behavior to be implemented in software. This is of great utility when, for example, a hardware device has failed but there is redundant information that allows the hardware to be simulated. Or when mass constraints dictate that a particular sensor cannot fly but that sensor's reading can be inferred from other spacecraft observations.

Having a drivers coordinate on the functionality axis in itself is a wonderful advance for future JPL missions. In pre-X2000 flight software architectures, the drivers were what attitude control systems (ACS) and ground-based sequencers commanded. And, the drivers directly mapped to a physical device and thus directly manipulated bits and bus addresses and memory maps. In these early architectures there was little of a generic interface and even less reuse. The core competency of JPL is in spacecraft systems and the software was reinvented time after time.

[Class of 'drivers' that are resources - each hardware device is obviously a precious resource as is power, etc.]

Commanders: Next up on the functionality axis are 'commanders' of the drivers. Examples are: reaction control, attitude knowledge, attitude control, time-based, open-loop sequencers, navigation and science.

Prior to DS1 {footnote-1 Prior to DS1 and subsequent to the demise of the DS1 'Remote Agent' - that is, aside from the short one year of on-board, high-level spacecraft autonomy.} the commanders level was the highest level within spacecraft flight software. Actually, the commanders level and examples described above is even slightly higher then tradition spacecraft because the navigation and science subsystems did not command. Whereas, in this X2000 design they do potentially (subject to a full-up analysis phase in the software development).

Configurers: The third level on the functionality axis are 'configurers' which reconfigure commanders to avoid resource constraints. Examples of configurers are: power and thermal subsystems. Configurers essentially dole our resources needed by commanders and might have limited reasoning capability to optimally, though locally, configure the resources.

[Commanders manage devices, Configurers manage resources?]

Automates: The highest level on the functionality axis are 'automates' which comprise the autonomy subsystems that close the commanding and configuration control-loops at the highest level. Examples are: planners/schedulers, mode identification and reconfiguration systems, etc. Like any software module described up to now, there is no distinction as to what is on-board, on-board in a companion processor, on another spacecraft or on the ground because the design space exists in the logical view, not the physical view. Of course, because 'automates' operate at the highest-level which is often (but not necessarily) the level with the lowest temporal completion constraints, it is possible for the 'automates' to be off-board.

Any given mission includes all of these functionality levels.

Hardware

The 'Hardware' axis has binary coordinates of 'present' and 'absent'; it is thus not quasi-continuous as is the functionality axis. The hardware axis embodies software models of physical-view devices; this is not to be confused with the hardware itself although the software models will be generally as close to the hardware as possible. Examples are software models for: SpectrumAstro Solar Electric Propulsion (SEP) Engine, DTI 1553 VME card, CT-401 Fixed-Head Star-Tracker, JPL General Purpose Board (GPB), etc.

The software models of hardware devices serve to implement the interfaces defined on the functionality axis. Several hardware devices might be required to implement a particular functionality model and a particular hardware device might implement several functionality models. This is a many-to-many relationship.

For a given mission, as hardware components are added, the software structure tends to be 'mirrored' about the functionality/hardware axis because the abstract functional models map to the hardware devices. This is not at all wasteful because, in brief, the functional models

are completely reusable and given the expected stability, the functional models can drive the standardization of spacecraft hardware. This point is revisited in depth later.

The 'Functionality-Hardware' plane is where the hardware people live. In particular, the 'drivers' coordinate on the functionality axis and the 'present' coordinate on the hardware axis is where hardware ought to live. 'Ought to live' because the drivers coordinate provides the idealized software model that hardware developers could strive for. This is thus unlike the 'present' hardware coordinate itself - wherein hardware developers produce software models that are drastically different from mission to mission. Prior to X2000, JPL mission find themselves in this second situation - an costly situation.

Faults

The 'faults' axis represents the departure from idealized software modules in terms of exceptions and includes recoveries in terms of exception handling. Like the 'hardware' axis, this axis has binary coordinates of 'present' and 'absent'. Examples of exceptions and exception handlers are: switch-stuck-off with handlers of switch-off, switch-on and warm-then-switch-on, etc.; image-buffer-full with handlers of discard-buffer, discard-image, flush-buffer/save-image, etc.; and others.

The 'functionality-faults' plane represents how spacecraft system engineers think abstractly about spacecraft - devoid of hardware, simulation and mission specifics. This plane has software models for everything that is generic about spacecraft systems - perhaps it embodies the content of a first coarse on spacecraft systems engineering. The plane is truly core-JPL and core-spacecraft engineering.

The 'functionality-faults-hardware' space represents the flight software; it is traditionally what flies on a particular mission. Note that this ignores the role of simulation software as flight software to backup failed hardware; this role is important but, perhaps aside from Cassini AACS, not employed often enough to increase spacecraft robustness.

Simulation

The 'Simulation' axis has quasi-continuous coordinates representing degree-of-simulation or fidelity. The zero coordinate on this axis has no simulation component. Any software module in the design space can be simulated and it is important to note that simulation should not be limited to hardware (and the environment).

The 'simulation-hardware' plane is where spacecraft hardware simulation generally takes place. The simulator interface is exactly as specified by the software model for the hardware device. The simulation often is performed remotely from the flight CPU because hardware devices are themselves located remotely and connected to the flight CPU via buses, like the 1553.

Simulation of the 'functionality-faults' plans at the 'drivers' coordinate plays an important role the software development process. Such a simulation is of the generic, lowest-level drivers - it allows all higher functionality to be developed and tested independently of specific hardware. When hardware becomes present, the software model for the hardware need only be used to implement the generic interface and then, once the generic interface is thoroughly tested with the hardware, the overall software is functional and tested.

Mission

The 'Mission' axis has coordinates representing each mission. For example: X2000, X2002, X2004, etc. In a sense, this is the single most important axis to ensure NASA's 'smaller, better, cheaper' mandate. This axis allows one to track, from mission to mission, the re-usability of the software.

If the software design is done well, the development for any particular mission ought to follow roughly the following course:

1) Select from a large number of predefined, implemented and tested software and hardware components (for example: switches, batteries, star trackers, etc.).

2) Commission new software modules for the new hardware devices (for example: the DS1 ion-propulsion-systems, a whiz-bang super-hi-resolution camera).

3) Extend existing higher-level functionality models has new technology is developed (for example: an improved planning engine, a computationally less intensive on-board navigation algorithm, or a new fuel-conserving RCS control mode).

4) Statically configure the selection of hardware and software components. The configuration includes component connectivity, mass, orientation, etc.

The selection of components is performed by spacecraft systems engineers in response to requirements specified by mission planners. The new software modules and the extending of existing models is designed and implemented by software engineers in consultation with domain experts. New technologies are developed and implemented by technology experts, generally prior to but often commissioned by a particular mission.

Development Approach

Given the flight software design space and the design guidelines detailed in the prior chapters, we are prepared to enumerate specific development approaches to ensure that the software requirements are met. We highlight six approaches: modularity, configurability, visibility, commandability, asynchronousity and dynamism. Each of these impacts one or more design guidelines and follows from the nature of spacecraft and their environment.

Modularity

Nothing has a bigger impact on a software development process than modularity.

[Impact on design, implementation, integration, testability and re-usability]

The 'Slice Model'

The 'Slice Model' in software design is the anti-thesis of a modular software design. The slice model was employed (and invented) out of necessity on DS1. Its need arose because of the view that autonomy experts in reactive, deliberative and reasoning systems needed to focus on their autonomy technology. Consequently flight software teams were composed based on technological expertise rather than based on modular 'domain units.' Without domain units there was no formal process by which to ensure that a particular domain was covered functionality. The result was, for example, perfectly functioning autonomy subsystems and no guarantee that the camera or propulsions system would function in closed-loop, top-to-bottom.

The 'slice model' has a horrible impact on integration. Each technology had a 'slice' of a particular domain unit, say the camera subsystem, along with 'slices' from telemetry, real-time and other components. Some of the components are of very high level (say at the level of mission goals) and some of the components are at a very low level (say at the level of flipping bits in a device driver). A functioning domain unit requires a small piece of each component - those pieces get merged during integration and thus only at the completion of integration will a domain unit even function for the first time. This integration process is actually what development should be; integration ought to be between domain units.

Although this problem was identified during the DS1 process, its impact throughout the development was vastly underestimated.

Impact of Autonomy

Autonomy has a significant impact on modularity. The impact arises because autonomy systems tend to employ a global view whereas modularity demands a local view. For example, model-based-reasoning systems for fault protection use inputs from numerous, isolated sensors, models of spacecraft and environment, and inference engines to infer some aspects of the spacecraft's state. By its global nature, autonomy poses a threat to modularity.

Modularity places significant requirements on autonomy. Autonomous systems can not be designed to usurp modularity even if it is at the expense of autonomy. Modularity is simple too important of a design principle to be abandoned for a relatively small (but admittedly important) goal of autonomy. This places the burden of conformance on the developers of autonomy.

The key to achieving modularity in light of autonomy is to split the autonomous system into several parts: a low level part that contains models and data; a high-level part with configurability, commandability, goals and global models; and a skew level with the inferencing engine. The low and high level parts populate the inferencing engine with models and thus the inferencing engine is functionally below both parts.

One specific example is that of 'inferred sensors.' An inferred sensor is a subclass of 'virtual sensor' (non-physical sensor) and thus of 'sensor' itself. Like every sensor an inferred sensor is an active object that asynchronously reports changes in the value of whatever the sensor purports to measure. These properties obviously make an inferred sensor indistinguishable from a sensor and thus plug compatible replacements for physical sensors. Any client of a sensor need not be concerned with from where or how the sensor's value is determined and thus 'inferred sensors' allowed the desired level of modularity.

Inferred sensors derive their state from global inferences performed over the entire spacecraft state (past and present) and based on the models resident in the inferencing engine. The effect is as desired: global inferencing provides an accurate value. The modularity is as required: no client need know the sensor's implementation. The implication is enormous.

Configurability

Like any fault tolerant, resource constrained system, spacecraft require unprecedented levels of configurability. (This precedence is based on the generally low levels of

configurability exhibited on spacecraft outside of ground-based intervention and attitude control systems.) Because of modularity requirements, the configurability must be provided locally, on a module-by-module basis. Three general areas of configurability are considered: fault protection, resources and structure.

Fault Protection

Configurability in fault protection allows for different exception handlers to be employed in response to an exception. The choice of an appropriate handler is generally based on many, many factors some of which are local, others of which are global, all of which vary in time. Configurability implies that a particular exception handler need not be statically chosen at design time but can be installed as desired.

Exceptions and exception handlers are defined locally. The exceptions are defined locally because only the module can know what exceptions are appropriate and from which exceptions the module cannot guarantee local recovery. The exception handlers are defined locally because only the module can know the context in which the exceptions happen and what actions are needed to recover. Note that handling an exception need not imply continuing from the exception but can instead imply aborting the computation and waiting for higher level aid. It is expected that a single exception might have many handlers; all the exceptions and their handlers are developed, tested and delivered as one.

Configurability in fault protection increases the robustness of spacecraft. Where appropriate exceptions get handled at the lowest possible level and allow activities to proceed in the face of uncertain conditions. Obviously, for some exceptions, it is not appropriate to handle the fault locally. Expect those exceptions to have an 'unable-to-handle' exception handler.

Resources

Configurability of resources allows for different methods of resource allocation to be employed in response to a resource request. Like exception handlers, resources can be allocated based on local or global considerations. Different allocation schemes might be appropriate at different times. Configurability implies that that basis for allocation need not be statically chosen.

Resources themselves have a multiplicity that influences the allocation basis. Some resources might be 'single' (a camera), or 'multiple' (disk files) or 'continuous' (power). It is usually apparent from the 'units' used in describing the resource which multiplicity it demands. For example: 'the camera,' 'a file' and 'watts'.

It is useful to think of resources as having a resource manager which serves as the broker for resource requests. A common example of a manager is a computer's 'memory manager' which maintains the memory subsystem by recording memory in use, free memory, available memory and which takes requests (via 'malloc()') for allocation of memory. Memory itself is a 'multiple' resource in which sharing of an individual unit may or may not be allowed.

Resource allocation can be of many types. The most common is for the resource to be used exclusively at the discretion of the resource user. There are other types: none, whereby any and all sharing is allowed; negotiated by, say, priority; or planned on a device and temporal basis. All these types have a place in spacecraft resource allocation.

Structure

Configurability of structure allows for different components to be employed in response to different requirements, faults or resource limitations. Examples are: replace a faulty physical gyroscope with an inferred sensor of precession; swap batteries on a cell failure; during encounter use a reaction control algorithm providing greater spacecraft stability at a cost of greater propellant use; use switch 'x' when requesting a power resource.

It should be noted that this level of structural configurability can be over-used by making everything modifiable. Such overuse is important to avoid because a clear definition of the static properties of spacecraft software has a tremendous impact on the design and ultimate performance of the software system.

Visibility

There is an inherent design tension between visibility and encapsulation. Design for encapsulation reduces interfaces by making as much as possible be an implementation issues; design for visibility expands interfaces by making more and more accessibly through the interface. Encapsulation is important as a general design principle; visibility is important when autonomy is involved.

Autonomous systems often have a deliberative component which bases decisions on models installed in an inferencing component. The models come from somewhere and that place needs to be the software model itself. That is, where possible and to the extent possible the software model must provide access to the information needed by the autonomous system. Providing access implies that the information must be both present and accessible. Keeping the information in the software model supports modularity.

One example of visibility is the requirement to have explicit, detailed state transition diagrams. State transition diagrams are an expression of the dynamic progression of a model; they are particularly important in engineered, real-time systems. Both nominal and fault behaviors are expressed in state transition diagrams.

Visibility to state transition diagrams provides, in large degree, the static models needed by autonomous systems for planning and fault protection. Planning concepts such as: timeline, state token, and action token map nicely to: active object, state and event. Similarly for fault protection where the sub-portion of a state transition diagram related to faults could populate the autonomy models.

Another visibility example is the need for power tables by flight software. Power tables are traditionally a design issue in which the power used by components and under what conditions is published in a tabular form. The table is used by ground sequencers to ensure that the power budget during a mission phase is not exceeded. The natural place for the power information is with the software model for the device which uses the power. The information needs to be used by the device when requesting power; it must be visible so that deliberative components can reason about the power demanded for a certain global activity.

Commandability

[nothing yet. What was I thinking. Callbacks?]

Asynchronousity

Spacecraft, by their nature and in spite of our best engineering efforts, are event-based, asynchronous systems. Faults occur and those faults are always asynchronous. Sensor states change based on actuation and those changes are asynchronous owing to various sources of indeterminacy. The environment is unpredictably rife with interesting scientific events like volcanic explosions and those events are always asynchronous.

Software modeling of this asynchronousity requires that the software be event-based, multiprocessing with a preemptive scheduler. Event-based ensures that external events map accurately into software events. Multiprocessing is a requirement because events arise independently and simultaneously from different sources. Preemptive scheduling acknowledges that some events are more important than others and that there cannot be one processor for each source of events.

Use of event-based, multiprocessing does not preclude time-based sequencing or periodic sampling implementations. The progression of time is itself an event, albeit a programmable one, in which a desired timing or periodicity is fully specifiable.

The distinction between multiprocessor and multiprocessing, preemptive scheduling is an implementation detail and ought not to influence the software design.

Dynamism

Spacecraft exist in a dynamic environment and are expected to perform different activities, derived from high-level goals, at different times. Examples of activities are launch detumble and checkout, thrusting cruise and encounter.

Software modeling of this dynamism requires that the software requirements not impose a priori limitations on the computational resources allocated to any particular module. Obviously computational resources are resources and need to be modeled as such and subject to the same requirements as other resources. However, attempts to preallocate memory or file space on a per module basis or to limit the number of processes/tasks runnable throughout a mission or to restrict the CPU available to a particular active object are largely unjustified. Doing such ignores well established technologies such as real-time, incremental garbage collectors and multiprocessor load-balancing algorithms for example.

Development Environment

Like the development approach, the development environment plays a major role to ensure that the software requirements are met in a timely and cost effective manner.

Project Database

The project needs a single, centralized, project controlled database. The database is to contain anything and everything related to the project. This ought to include, minimally: 1) Project Requirements and Finances 2) Staff and Organization Charts 3) Mission Design 4) Hardware Design 5) Software Design, Environment and Implementation 6) Documentation, WEB and otherwise 7) Science and Engineering Data Results. The database and the hardware on which it runs become the legacy of the project. All future access to the science data should be accessible via the database (but not to exclude other

distribution media like CDROMS). The WEB pages (public and private) exist for all time. The tools used for development and testing exist for all time. The monthly schedules and cost estimates and progress reports and up-scope analyses exist for all time.

The project database appears in direct opposition to the entrenched processes. A project passes through the PDC, the Design Hub, the FST and the Ground data systems and along the way leaves a bit of itself in each of the waypoints. These waypoints have a sense of control; but none of consultations nor delivery. This entrenched process is inverted. The waypoints need to deliver tools and expertise and documentation to the project and they need to interface their databases with the projects database.

These waypoints are like eat-in-only restaurants: you come in, you eat and leave; you've got a memory, perhaps a buzz, perhaps heartburn; there are no doggy-bags, no take out, no delivery.

A project database is a first step towards a paper-less design. If hardware CAD/CAM systems eventually are integrated with the database, then that would be one large step towards ensuring that hardware and software models are synchronized and that data needed for the required software model visibility is accurate.

Uniform, Stable Tools

The reality of software development is that developers work in remote locations, on distinct networks and files systems, to different daily time schedules and with different backgrounds. Add to this mix a project with multiple hardware target platforms and software tools distinct at all locations and prone to unannounced version update and you have a disaster. A disaster not unlike others of recent note.

One requirement for the development is that it be uniform across all machines routinely used for development, have a minimal set of target platforms (ideally a single platform) and be unchanged beyond month one. This is not an ideal situation; this is the definition of an environment (albeit a static one). If a project finds that time is spent, beyond month one, on compilers and version control systems and file system organizations then that spacecraft project is doing development outside of core NASA and JPL competency.

The apparent hardware platform of choice for spacecraft appears to be VME cages with multiple single-board-computers each running VxWorks. This choice is well beyond the previous state of having a custom operating system (OS) on a mission-by-mission basis and beyond the promises of some future OS that will unify the universe. If VME CPUs running VxWorks is the flight configuration (and if JPL's core competency is spacecraft) then the use of VxWorks boxes ought to be promulgated loudly and widely.

Note that the software architecture presented in this document makes clear the distinction between logical view and physical view. The selection of a processor and OS belongs to the physical view and is thus a relatively minor component in a spacecraft software architecture.

[Software specifics]

Having uniform, stable tools is an important prerequisite to having developers work with one mind-set and share one experience. As the number of platforms and software tools increases so does the ability of any one person to fathom all the details of each platform.

The result is subgroups of developers that know a lot about a few things and a little about other things. The lose of one group jeopardizes the entire project - in the extreme the lose of a single individual can sink the project.

The value of a shared development experience and its facilitation by limiting the hardware and software options cannot be under-emphasized.

Development Process

[More on specific phases... at least: requirements, analysis, design, implement, test]

The development process puts the software design and development together into a manageable entity. With the process come the generic milestones, the staffing requirements in number and expertise, the documentation framework, the organization chart, etc.

Language Independent

It is important that the process be suitable for different software implementation languages. Specialization up front on C++ or Java or CLOS is an indication that the modeling phase is not accurately modeling the domain because obviously the spacecraft domain is independent of the implementation language.

The implementation language even need not be properly object-oriented. Clearly spacecraft have design requirements and platform constraints within which object-oriented languages might not fit. Again, modeling is what is important, not the implementation language.

Iterative, Incremental

The development process needs to be iterative and incremental (see "Developing Object-Oriented Software..." (DOOS)). Such a process is 'incremental by requirements' in that each increment provides additional functionality to satisfy the customer's requirements. And 'iterative' in that each increment includes some rework. Such a 'strongly incremental' process is appropriate when: 1) requirements are uncertain and complete, 2) technologies are use with which the development are unfamiliar, and 3) project is complex. Again, see DOOS for the above as one example.

The process needs to be understood by all developers and all developers must sign on the process. Any group that, during the development process, concludes that the process is inappropriate for them needs to be reconstructed or replaced. The middle of a process is not the appropriate time to debate the process.

Team Composition

Software development needs to be performed by software development experts; not by people knowledgeable in spacecraft technologies. Such development experts will be comfortable with an object-oriented, incremental, iterative development process, with designs based on reuse, modularity and encapsulation and with the value of a stable uniform development environment.

Ideally the spacecraft technologists would have the expertise in software development but it is unreasonable to expect that a-priori. After all, 'NAV' people need to be knowledgeable

about stellar image processing and triangulation and 'ACS' people need to know about control theory. Without the expertise in software development, the appropriate role for a technologies is as a 'domain expert' with important roles in the software analysis and design phases of the software development process.

Conclusions

[Summarize link between 'design guidelines' and responses.]

11.2.6 Functionality

11.2.7 Verification (and Testing)

11.2.8 Implication for Hardware Architecture

11.2.9 Project Issues

11.2.10 Conclusions

11.3 Plug-and-Play Architecture

[This is two separate submittals. No attempt has been made yet to merge them.]

11.3.1 Human organization

Software development is a human endeavor, and it is important to recognize that the information flow among the humans writing the software may or may not reflect the structure of the information flow among the components of the software. Ensuring that information flows properly between the software developers is at least as important as ensuring the same for the software components, and perhaps more so because human communications channels are noisy and bandwidth-limited. The design of the software team should be an integral part of the design of the software structure.

For example, consider the software needed to control a device. This software can be written by the person or team that designed the device, but this team is likely to have more training and experience in designing devices than programming. Alternatively the software can be written by a trained programmer who is not initially familiar with the operation of the device. It would be advantageous if the software can be structured in such a way that the device designer can write the device-specific software with less effort than it would take to transfer the required knowledge to a programmer.

Realizing the vision of a completely plug-and-play architecture where device designers write the code for their devices, scientists write the code to control their experiments, spacecraft designers write down the flight rules for the spacecraft, and everything comes together automatically, requires some radical rethinking about what it means to write software. It is almost certainly the case that to realize this vision will require languages capable of expressing the kinds of information that humans exchange when designing software systems, including models, priorities, intentions, and reasons, not just procedures. This is a very long-term vision. In the meantime, simply recognizing that information flow among humans is an essential part of making a spacecraft work is an important first step. Too often architectures are designed and block diagrams are drawn without taking this into account. The result is endless series of meetings, memoranda, misunderstandings, and missed deadlines.

11.3.2 Modularity

A spacecraft software architecture should support modularity and reusability. Ideally, the architecture should support full plug-and-play capability, where every piece of hardware is delivered as a package with all the software needed to run it. To caricature the vision somewhat, every piece of hardware comes with a floppy disk. You collect all the software on the disks, throw them into a magic black box, and out comes a complete set of spacecraft software ready to run. This scenario raises three questions: 1) what needs to be on the floppy disk in order to make this work, 2) what is inside the "black box" and 3) what price are we willing to pay in terms of up-front effort, increased risk, and reduced efficiency in order to realize this vision?

To answer these questions, consider the following scenario. We have a spacecraft that includes a camera. This camera has two connections to the rest of the spacecraft, a 1553 bus client and a power connection. The 1553 port is connected to a non-redundant 1553 bus, and the power terminal is connected to a non-redundant power distribution unit, which is also connected to the 1553 bus. There is also a CPU and a 1553 bus controller in a VME cage. Each of these components comes with its own plug-and-play floppy disk.

Now consider the process of turning on the camera purely from the point of view of the hardware. The CPU sends a command over the VME bus to the 1554 bus controller, which sends a command to the PDU, which changes the state of a switch, which causes electrical power to be supplied to the camera.

There are two important points to note about this scenario. First, the camera (hardware) does not participate at all in this process. Turning on the camera is a side-effect of a state-change in the PDU, which is a side-effect of a command sent over the 1553 bus, which is a side-effect of a command sent over the VME bus, which is a side-effect of a computation performed on the CPU. Thus, if we suppose that the camera's floppy disk contains executable code we are forced to conclude that this code alone is not enough to turn the camera on. Either the camera code has to interact with code elsewhere in the system, or there must be something else on the disk.

Let us examine the first possibility, that the disk contains a "camera software object" that negotiates with other software objects in the system to turn on the physical camera. This collaborate-object model is very popular, but not often fleshed out beyond a vague block diagram whose link semantics are not well defined. In the absence of a concrete proposal from its advocates, I will set up a straw man for what such a negotiation might look like:

1) Mission manager to camera: turn yourself on
2) Camera to power manager: what is my power port connected to?
3) Power manager to camera: PDU1, port A
4) Camera to PDU1: Please turn on the power at port A
5) PDU1 to data bus manager: which bus controller is master to my 1553 bus connection?
6) Data bus manager: 1553 bus controller 1
7) PDU1 to 1553 bus controller 1: Please send the PDU1_PORTA_ON command
8) 1553 bus controller 1 to VME manager: what is my VME I/O address?
9) VME manager: 0x1234

   [1553 bus controller object twiddles bits at 0x1234 which sets in motion the chain of events that turns on the camera.]

There are several things to notice about this dialog. First, it is pretty complicated. Nine message transactions were required. Some of these transactions could be done once at system initialization and cached, but in general they cannot be. If there are redundant PDU's, for example, then the camera has to ask which one to use every single time.

Second, this is an absolutely trivial example. It is an open-loop change of a single binary state with no fault management, no redundancy, no conflict detection or resolution, no constraint checking or recovery.

Third, there is this hypothetical "power manager" object that is not associated with any particular piece of hardware[i]. It is thus presumably part of the "black box" infrastructure of the plug-and-play architecture, and is designed differently from the hardware-specific objects. This distinction is evident on some block-diagrams of negotiating objects (e.g. by having dramatically more lines coming out of them) but are otherwise undistinguished. This leads to the impression that all these objects have a homogeneous structure, which would appear unwarranted at best, and a serious mistake at worst.

It is incumbent upon advocates of the negotiating-object model of modularity to provide detailed descriptions of exactly how the negotiations happen. What are the data structures that are passed around? How are failures handled? How are decisions modulated by the global mission state? What is the API (abstract programmer interface)?

This brings us to the second possibility, that there is something on the camera floppy disk besides executable code. One possibility is that the floppy disk includes object models that are interpreted by a run-time engine (or set of engines) such as the New Millennium Remote Agent, or similar system. This approach has the advantage of having been implemented and demonstrated already, and so there is a very detailed (and long) story that can be told about exactly how this option works. This story is beyond the scope of this document, and there is no doubt room for improvement. For example, the current Remote Agent design uses three different and redundant representations. A unified representation would be very useful.

Finally, we observe that the fundamental problem in designing a plug-and-play architecture for spacecraft is the many different ways in which spacecraft subsystems can interact. The list of interaction mechanisms found elsewhere in this document may be a useful starting point for the design of a unified representation for hardware objects. For example, the camera floppy disk might contain information such as the following:

- I am a camera, which is a powered-bus-device
- When my power-state state vector component has the value on I draw seven watts of power.
- My bus-interface is of type 1553-bus-client
- I have an additional state-vector element called picture-state which can take on the values off, warming-up, ready, and taking-picture.
- Whenever my power-state is off, my picture-state is also off.
- When my power-state transitions to on, my picture-state becomes warming-up for five minutes, and then becomes ready.
- When my picture-state is ready I can respond to the following additional method: take-picture, which has the following effects...

This object model can be considered a "program" for a "black box" similar to the remote agent that has planning, execution, and fault detection and recovery capabilities built in as infrastructure.

---

[i] This is because it must be able to inform any piece of hardware of its power port the design of the power distribution system. In fact, the straw man example makes assumption that all power ports are connected to one or more PDU's with designated switched. To really get this right would make the example even more complex than

11.4 Uplink System Design / Command & Control Capabilities

3.1 New Command & Control Process

Rather than the traditional single-mission design approach, the X2000 software architecture, uplink process, and command & control capabilities are intended to support a series of missions characterized by development and launches spread out over many years, long mission flight times, a variety of different science payloads, and at least a decade's worth of evolving advanced hardware and software technology.

The X2000 software design approach taken to accommodate multiple missions and evolving technology is to assume a spacecraft avionics design and shared ground data system common to all missions. Furthermore, flight operations and other post-launch cost considerations have led toward the design goal of developing a new command & control process that is compatible with very low ops team staffing levels.

This new process places less emphasis (and manpower) associated with development on the ground of a constraint-checked, conflict free, predictive-model based, timed sequence. Rather, the X2000 uplink process design goal is for a process that allows an on-board "stack" of multiple, time-windowed, prioritized command macros that may be time or event triggered. This approach not only saves the enormous costs of detailed sequence planning and the development and running of highly accurate predictive performance models, it enables command and control of on-board processes that are inherently unpredictable, such as anomaly response (fault protection), variable rate data compression, science opportunity detection, and even intelligent agent command generation. S/C performance efficiencies will result from triggering commands using on-board actuals, rather than ground model predicts.

An important feature of the new uplink process is that it can be operated in a lower efficiency mode by defaulting back to the old ground performance- model based, timed command sequence mode. This mode on X2000, however, will depend on much coarser (e.g., lower cost) performance and resource management models than have traditionally been developed and maintained by flight projects like Galileo and Cassini.

A second important feature of the new uplink process is that it provides on-board constraint checking that is a migrated version of similar ground based constraint checks. This permits the same conflict identification process of macro calls to be run first on the ground using coarse model based constraint checks, and later, on the s/c using s/c actuals. Runs on the s/c will allow new, event driven on-board command macro calls to originate and be entered into the command macro stack.

The third major feature of the new uplink process is a simple priority based conflict resolution process, that resolves conflicts between macros by scheduling the macro with the highest priority. This capability when used operationally by undersubscribing high priority macros and oversubscribing lower priority macros, is expected to provide "guaranteed" events together with a highly efficient supplement of "bonus" events. Undersubscription margins will allow for late entry of on-board event driven macro calls, some of which may be high priority, and some of which will be lower priority. Likewise,

late, on-board intelligent agent or "smart" planner originated macro calls may be pre-assigned high or low priorities based on mission experience, "trust", and confidence in that agent.

FIGURE 3.1 X2000 COMMAND ARCHITECTURE

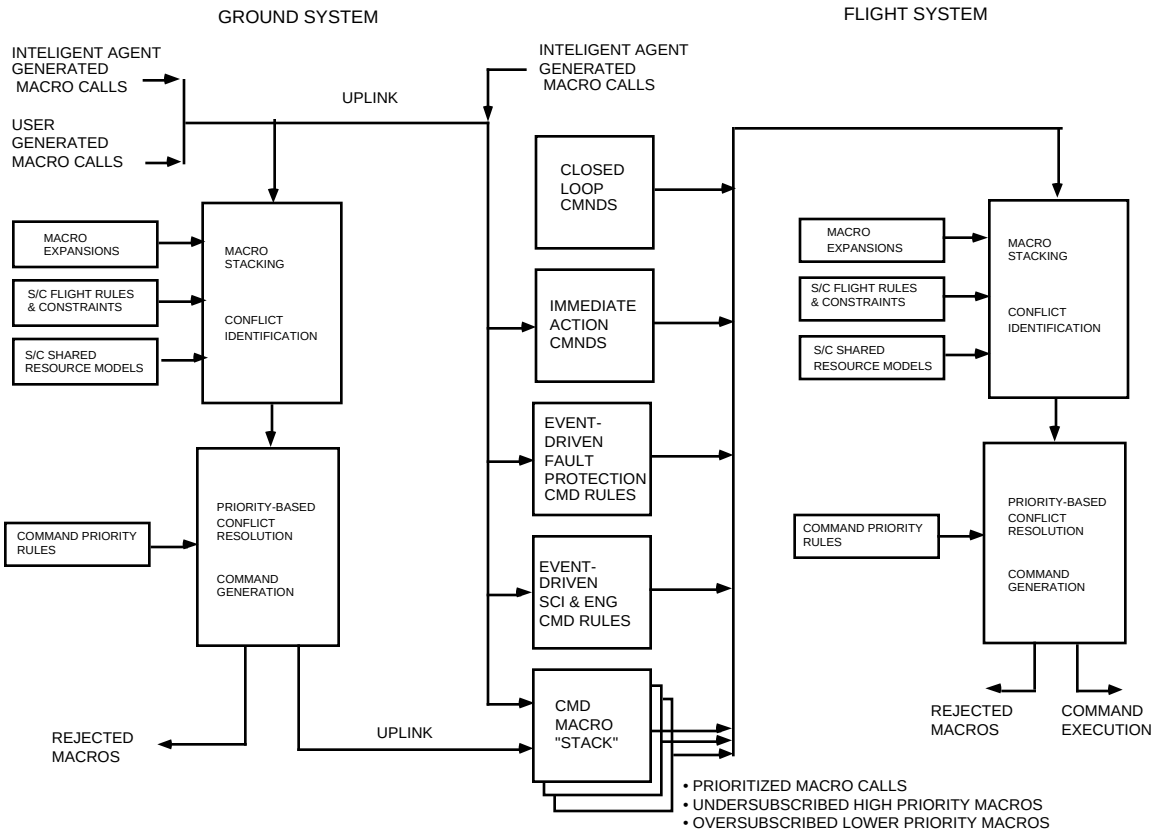


Figure 3.1 is a block diagram of a flight and ground command architecture that enjoys the features mentioned above. User generated command macro calls can be input to the system, run through the ground conflict id and resolution process and get uplinked as an integrated set of macro calls, or the ground process can be bypassed and macro calls can be uplinked directly to the on board macro "stack". The on-board conflict id and resolution process accepts inputs from the macro stack as well as inputs from the "boxes" above, including selected on-board closed loop processes, uplinked "real-time" immediate action commands, fault protection commands, and event driven science and engineering commands. All of these command sources have pre-agreed priority assignments to enable simple priority-based conflict resolution. Macro calls that specify large time windows will have a better chance of getting executed than macros that must be executed at one specified time. Finally, Figure 3.1 shows that macro calls can originate from intelligent agent software located either on the ground or in the spacecraft.

FIGURE 3.2 TRADITIONAL COMMAND ARCHITECTURE

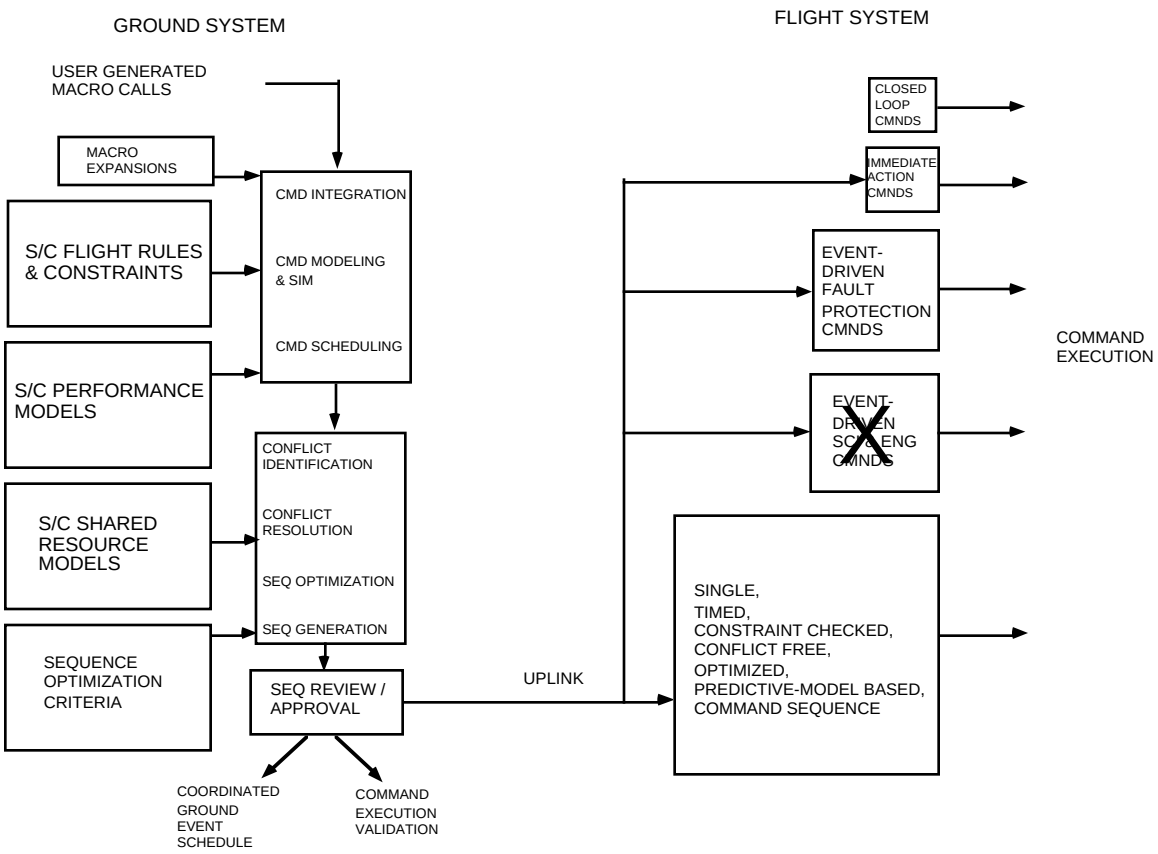


Figure 3.2 is a block diagram of the traditional command process architecture for comparison with the new design. One of the obvious differences is the fact that there is no on-board constraint checking and conflict resolution process equivalent to the ground process. The traditional ground process places much more emphasis (and $) dealing with flight rules and constraints, precise / detailed predictive s/c performance models, and sequence optimization. The product uplinked is a single, timed, constraint checked, conflict free, optimized, predictive-model based command sequence. It is incompatible with unpredicted events (such as fault recovery commands) and typically “crashes” if an unpredicted event occurs. For this reason it doesn’t allow event driven science and engineering events (the box with the “X” over it). It doesn’t accommodate late command entry from on-board intelligent agents.

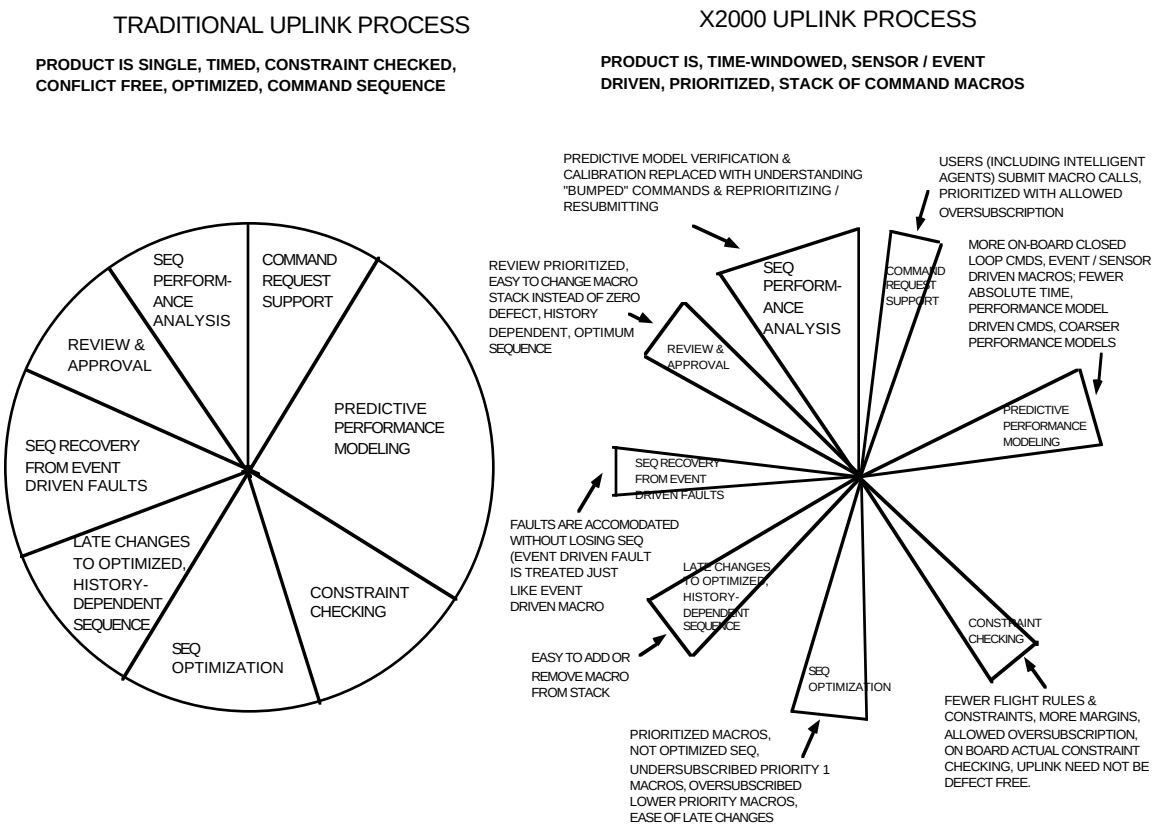


FIGURE 3.3 COMPARISON OF TRADITIONAL VS. X2000 UPLINK ACTIVITY EFFORTS

Figure 3.3 is a pie chart of how uplink resources are spent operating the traditional sequence uplink process and how these operational resources are reduced by the new X2000 process. The new process will offer major cost reductions associated with the development, calibration, and repeated running of predictive performance models. For example, instead of using a detailed performance model to predict slew time and settling time, on-board sensor actuals will trigger the next event. Instead of a detailed, carefully calibrated thrust-time vs. delta V model, recalibrated for changing s/c mass during the mission as propellant mass is depleted, an accelerometer will trigger the thruster-off command. Figure 3.3 explains for each major uplink task, how the new command & control design will enable significant operations cost savings.

Constraint checking will be simplified first, because the X2000 s/c will be designed for operability with minimum flight rules to constrain operations and with adequate performance margins such that the interaction between command macros competing for shared resources will be minimized. Second, constraint checking effort will be reduced due to the new uplink process since this permits oversubscription and doesn't require creation of a conflict free sequence. Finally, constraint checking on the ground can be performed with "looser" tolerences and less precise models, since constraint checks will be reperformed at a later time on board the s/c using parameter actuals, rather than ground model predicts.

Sequence optimization is eliminated in the new uplink process and replaced instead by simpler macro prioritization.

Late changes to an optimized, conflict free, history dependent, timed sequence are complicated to implement. For instance, adding or removing a slew in the middle of a sequence will cause all subsequent slew time predicts to change along with the times of subsequent events, and the sequence will have to be redone. In the new X2000 process, adding or removing a slew is just a matter of adding or removing a macro from the prioritized stack.

The same process above that accommodates user initiated changes, will accommodate changes initiated by on-board faults and failure recovery macros. Thus the new X2000 command process will continue to execute its prioritized stack of macros following fault recovery, while in most cases, the traditional, timed, history dependent sequence will abort, fall back to some safe state, and wait for ground reconstruction, fault modeling, and sequence redesign, before it can resume.

Instead of reviewing a sequence about to be uploaded to verify that it is optimum and conflict free, the new uplink process simply looks at priorities and probably margins associated with the successive release of shared resources (e.g., 50% of predicted available memory to priority 1 macros, 30% to priority 2, 30% to priority 3, etc.)

Performance analysis may be the one uplink task area that doesn't go down (the pie wedge in figure 3.3 stays big), although it substantially changes character. Instead of analyzing sequence execution to see if s/c performance actuals met s/c performance predicts and for data to calibrate and verify predictive models, the X2000 command process will involve analyzing command actuals to see which commands got executed and which ones didn't and try to understand why and decide whether to reprioritize / resubmit "bumped" commands.

TABLE 3.1: X2000 COMMAND ARCHITECTURE: 10 MOST IMPORTANT

1. TRIGGER ENGINEERING & SCIENCE COMMAND MACROS BASED ON ON-BOARD EVENTS / SENSOR STATES RATHER THAN PREDICTIVE PERFORMANCE MODELS.
2. FLY ON-BOARD CONSTRAINT CHECKING AND CONFLICT RESOLUTION SW SIMILAR TO GROUND SYSTEM.
3. USE SIMPLE COMMAND PRIORITY SCHEME FOR CONFLICT RESOLUTION.
4. UNDERSUBSCRIBE HIGH PRIORITY COMMANDS (GUARENTEED) AND OVERSUBSCRIBE LOW PRIORITY COMMANDS (BONUS).
5. DESIGN S/C FOR OPERABILITY WITH MINIMUM NUMBER OF FLIGHT RULES & CONSTRAINTS.
6. ACCOMODATE FAULTS AND UNPREDICTABLE EVENTS WITHOUT LOSING "SEQUENCE" (MACRO STACK).
7. PROVIDE A DUAL FLIGHT-GROUND COMMAND PROCESS ARCHITECTURE SO FUNCTIONS MIGRATE EASILY.
8. PROVIDE FOR LESS EFFICIENT, GROUND BASED, CONFLICT FREE, TIMED SEQUENCE COMMAND MODE AS A FALLBACK CAPABILITY.
9. EXPLOIT BOTH GROUND AND FLIGHT INTELLIGENT AGENTS AS SOURCE OF PRIORITIZED COMMAND MACROS.
10. MINIMIZE REQUIREMENTS FOR COORDINATED FLIGHT & GROUND EVENTS (BECAUSE MANY S/C ACTIONS WILL BE UNPREDICTED).

A summary of the principles of the new X2000 command and control process described above is provided in Table 3.1. This lists the top 10 steps to be taken to move from the traditional ground sequencing process to the new X2000 command and control design.

11.5 Information Systems Architecture

"A Database is a treaty that governs the behavior of Users"

Dr. Paul Gorham

University of S. Wales

1.0 Introduction

This section of the X2000 Unified Flight Ground Architecture Document address the Information System Architecture, specifically as it relates to X2000 databases. The X2000 database presents new opportunities and challenges for information capture and retrieval. The ultimate goal of this architecture is to reduce the cost of:

- applications development
- ad-hoc query processing & data retrieval
- pre-planned (server push) data transfers
- replication of distributed data
- roll-up of distributed information necessary for centralized decision making.

The two fundamental systems architectural principles will assume a centralized information base that is distributed and object-oriented in structure. Where applicable, industry standards will be used as a basis for some decisions with respect to implementation.

2.0 Information Architecture

Timely, correct, and properly formatted information is necessary for timely decisions at all phases of the mission life cycle. This architecture will attempt to cover the mission life cycle as well as provide data to subsequent X2000 engineering efforts.

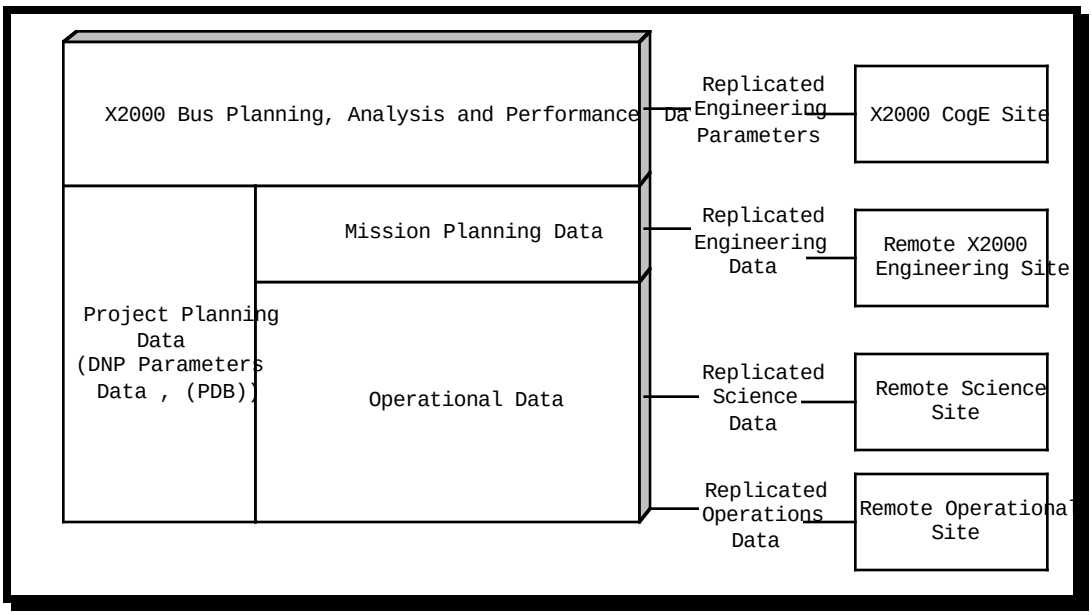


Figure 1 - High Level Information Architecture

Figure 1 illustrates the highest level of X2000 information architecture. Information is decomposed into 4 areas that reflect overall mission or project state. Applicable data is replicated from the centralized repository to remote sites via database replication operations.

Project Design information is very high level information about the actual Ice & Fire mission and vehicle configuration in question. This structure is maintained in a Parameters Database that will be provided by the DNP organization[ii].

X2000 Bus Engineering Data contains data relevant to the generic X2000 hardware configuration and performance. The ultimate goal of this information is to provide input for X2000 Bus process improvement and provide planning data for subsequent X2000 missions.

The Mission Planning structure will be a repository for sub-system engineers to populate data structures with relevant configuration and calibration information about their respective sub-systems' sensors and effectors.

The Operational Data repository will contain data that will support both real time, non-real time, and science payload operations of the X2000 mission. It will also contain a replicated image of the spacecraft flight database.

Project cost reductions will be achieved by centralizing all X2000/Ice & Fire databases. The cost savings will be made by centralized Database Administration staff and functions (e.g. Security, Configuration Management, back-up, Recovery, and maintenance activities) can be located in a single facility. This will eliminate redundant database administration activities at distributed sites. X2000 will take maximum advantage of distributed database

[ii] The current DNP Parameters Database (PDB) is a relational structure which provide design tools in different phases of design. This database supports data archiving

operations. Specifically, the centrally maintained X2000 data will be replicated to remote sites and remote data products originating from remote sites will be rolled up into the central X2000/Ice & Fire database.

2.0.1 Mission Planning and Engineering Base

This information structure will be defined by the RDL syntax. RDL is an emerging industry standard syntax used for the definition of the Project Command and Telemetry Database. The basic idea for X2000 will be *the specification derives both information and classes* to manipulate the mission information. The RDL syntax can support both TeleCommand and Telemetry specifications set forth by the Consultative Committee Space Data Systems (CCSDS). The following is an example of RDL syntax to describe the CCSDS Header of a Telemetry Packet:

```
PACKET P002 APID=02, DESC="Pri S/C Processor Fast Normal Mode Packet", STALE=148
      RECORD CCSDS_Header APPEND,DESC="CCSDS Header"
        UNION HDR1     DESC="CCSDS Header 1st 16 bits"
          ui     pvno    mask=%b1110000000000000, lshift=-13, desc="Packet Version Num.
Bits  0 - 2"
          ui       pckt       mask=%b0001000000000000, lshift=-12, desc="Packet Type
Bit   3"
          ui    shdf    mask=%b0000100000000000, lshift=-11, desc="Secondary Header Flag
Bit   4"
          ui     id          mask=%b0000011111111111, lshift=  0,   desc="Application ID
Bits 5 - 15"
         END
         UNION HDR2     DESC="CCSDS Header 2nd 16 bits"
          ui        segf    mask=%b1100000000000000,   lshift=-14, desc="Segment  Flags
Bits 0 - 1"
          ui    scnt  mask=%b0011111111111111, lshift=  0, desc="Source Sequence Count
Bits  2 - 15"
         END
         UI      plen        desc="Packet Length"
        MET   stime      desc="Secondary Header Time (64 bits)"
       END
```

The above example shows the flexibility of the grammar. It is envisioned the language would constitute the basis of parsers that would read the Mission Planning Command & Telemetry database and would automatically generate both real time commanding database, telemetry database, and the classes/objects that would manipulate these information bases respectively. The following is a high level architecture of the Mission Planning Information Architecture.

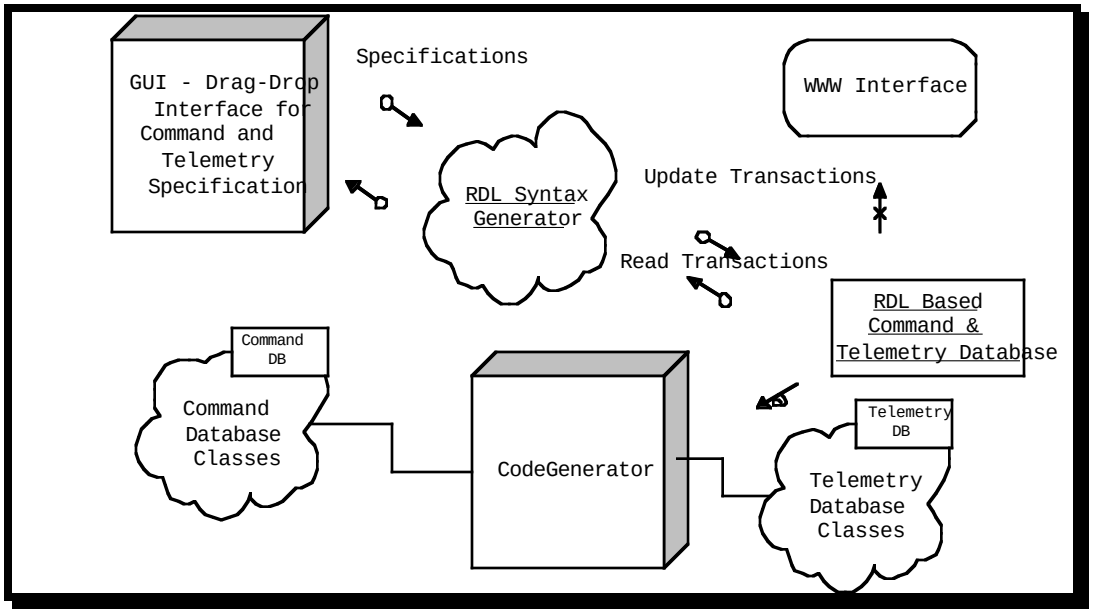


Figure 2- Mission Planning Information Architecture

The specification phase would involve the population of the Project Command & Telemetry Database. The human-machine interface would extend an easy to use drag-drop interface for enumeration of spacecraft hardware actuators, sensors, derived items, etc. by engineers, scientists, and mission planners. Specific information fields would be handled by pop-up Property Pages with context sensitive help to aid the operator in creation of a complete specification. The back-end of the interface will take the graphic specification of the command or telemetry point and generate the RDL syntax necessary for population of the database. The database would extend interfaces for applications to query existing specifications. This interface would be implemented through application, World Wide Web (WWW) applets, and ad-hoc interfaces.

To the greatest extent possible, the systems and systems generated from specifications will be data driven in nature. The concept of data driven means there is some object that contains information about how to process other objects. The algorithm/program becomes invariant—only the data is updated. Software maintenance becomes a database update..

All applications that access X2000 information will be created to support the Unicode standard and designed so application resources (Menus, Dialogs, Strings, etc.…) can be placed in "resource-only" dynamic link libraries (DLL). This will facilitate deployment of the applications into the language of our International Science & Engineering Partners and Users. (Applications will be capable of being "hot-switched" with indigenous language prompts, menus, and dialogs.)

The back end of the Mission Planning System shall support the automatic generation of the real time command & telemetry databases.

2.0.2 X2000 Bus Planning, Analysis and Performance Base

The X2000 Planning, Analysis and Performance information base will initially become a repository for planned performance data for the X2000 bus. Ultimately, this repository will be populated with real time performance data and will maintain metrics and analysis data on planned versus operational performance of the X2000 bus. It is envisioned that the data from this system will constitute the information content of the feedback loop for mission designers of subsequent X2000 missions. The substance of this data can include (but is not limited to):

- X2000 system, sub-system, sensor, actuator processing abilities and limitations
- Calibration Data[1] (Pre-Launch calibration planning, Post Launch updates based on performance)
- Consumable Data (Propellants, Battery Cells, etc.) Planned and actual data
- Engineering Data[1] (raw and summarized)

It is expected that this data will be utilized for planning subsequent X2000 missions. This database can be thought of as a process improvement repository. The goal of this information is to improve the quality of future X2000 missions and reduce their cost.

2.0.3 Operational Mission Base

This aspect of the X2000 information base is intended to maintain real time operational data for both X2000 bus systems, C&T, orbital/tracking/navigation/attitude, engineering and science payload. The Commanding and Telemetry databases are created from specifications in the Mission Planning Phase.

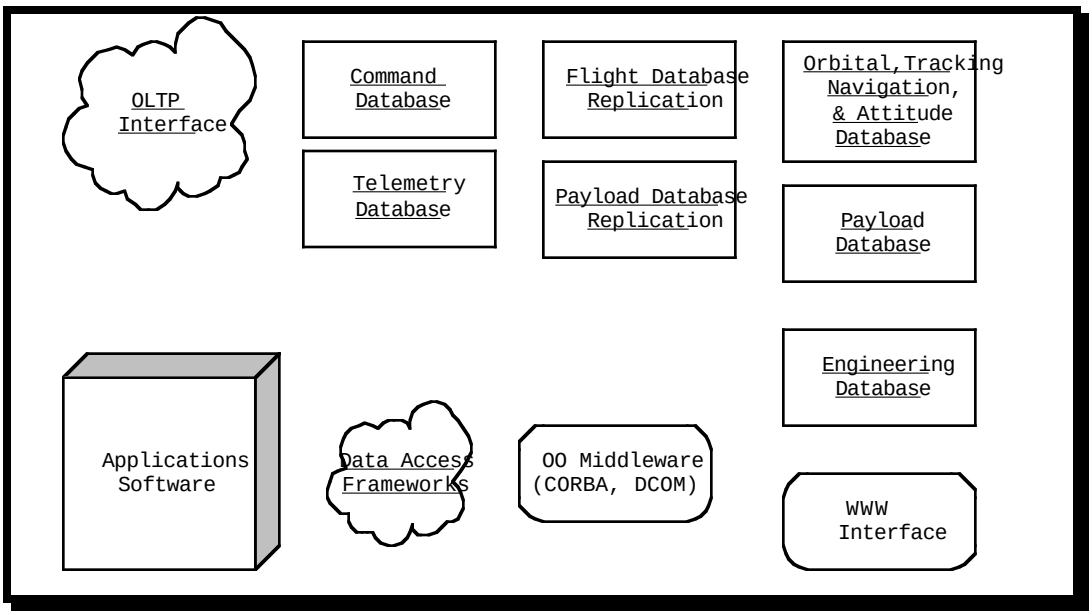


Figure 3 - Operational Database Architecture

The focus of this document is not to design the schema for these repositories. The operational data base will be contained within an OODBMS. Access to the database will be made through an Object Oriented middleware (CORBA or DCOM). The actual requests from these interfaces will be handled be Transaction Processing (TP) monitor. This class

of software is being used because the ultimate volume of data for 4 missions will be sufficient to crash a system without a transaction monitor.

3.0 Flight Database

The vision for the flight database is to have an OO client/server architecture.

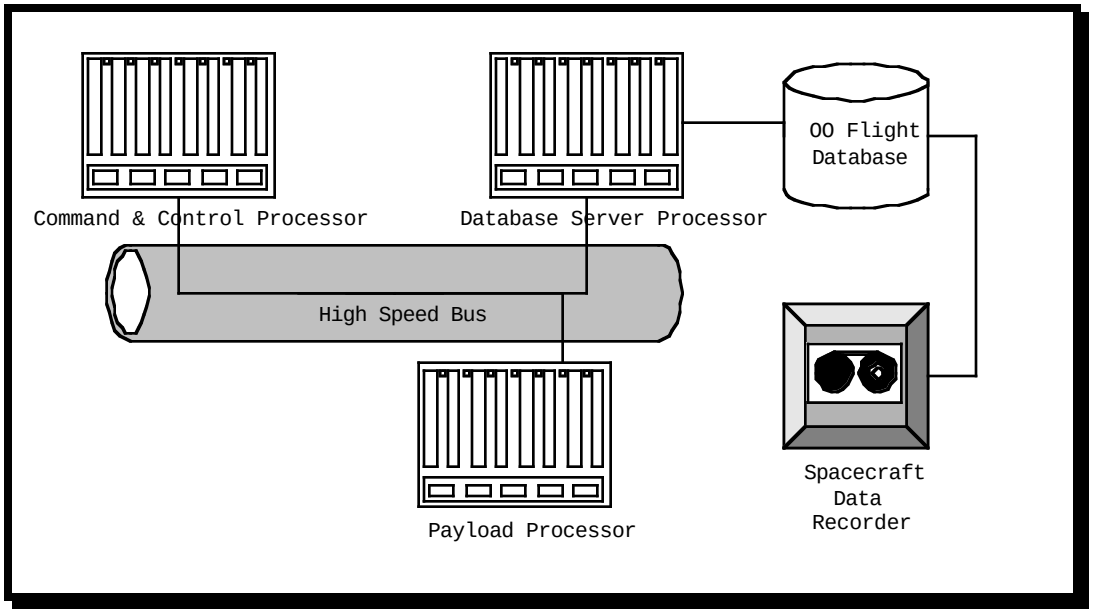


Figure 4 - Flight Database Configuration

All applications will retrieve & post their information to the centralized database. For time critical high performance applications, portions of the database could be replicated to other processors. The database would maintain a non-volatile copy within the Spacecraft Data Recorder. Applications in the flight system will access data by sending messages to the database server. This messaging paradigm will follow the Law of Demeter.

11.6 Scaleable & Flexible Sequence

11.6.1 Introduction:

Supporting future JPL missions has become a very big challenge. These days, for instance, the pressure's on having a visionary use of technology, being adaptable to a range of S/C and missions, being able to shorten the development time, and having a smaller flight team[1]. Event driven sequencing, priority based sequencing, sequence over-subscription, and on-board sequence restart after S/C fault recovery are another added challenges to be met[2]. Yet, important these requirements are, they can't cost too much. Such expanded requirements to S/C, mission operations, and flight ground S/W development have clearly become the central success to a JPL faster, better and cheaper mission.

These new considerations have begun to drive the TMOD MP&A's decision about how to meet current and future JPL missions' objectives. The search is on for the solutions that offer: unified flight ground S/W architecture, incremental implementations, and an end to traditional sequence obsolescence**. Scaleable and Flexible Sequencing** is a recommended sequence scheme to meet the about-mentioned needs.

[1] X2000 S/W architecture programmatic requirement, presentation handout, 5/27/97, Robert Barry

[2] Ice & Fire Command Architecture: The Goal, 4/23/97, John Carraway

11.6.2 Incremental Implementations:

It's pretty hard to re-engineer sequence constructs from the traditional time based sequencing to a fully automated on-board sequencing approach. Therefore, effective re-engineering solutions need to enable gradual, piece-meal adoption.

11.6.3 **Phase 1: Time Based Sequence** (see fig. 1 time based sequence example)

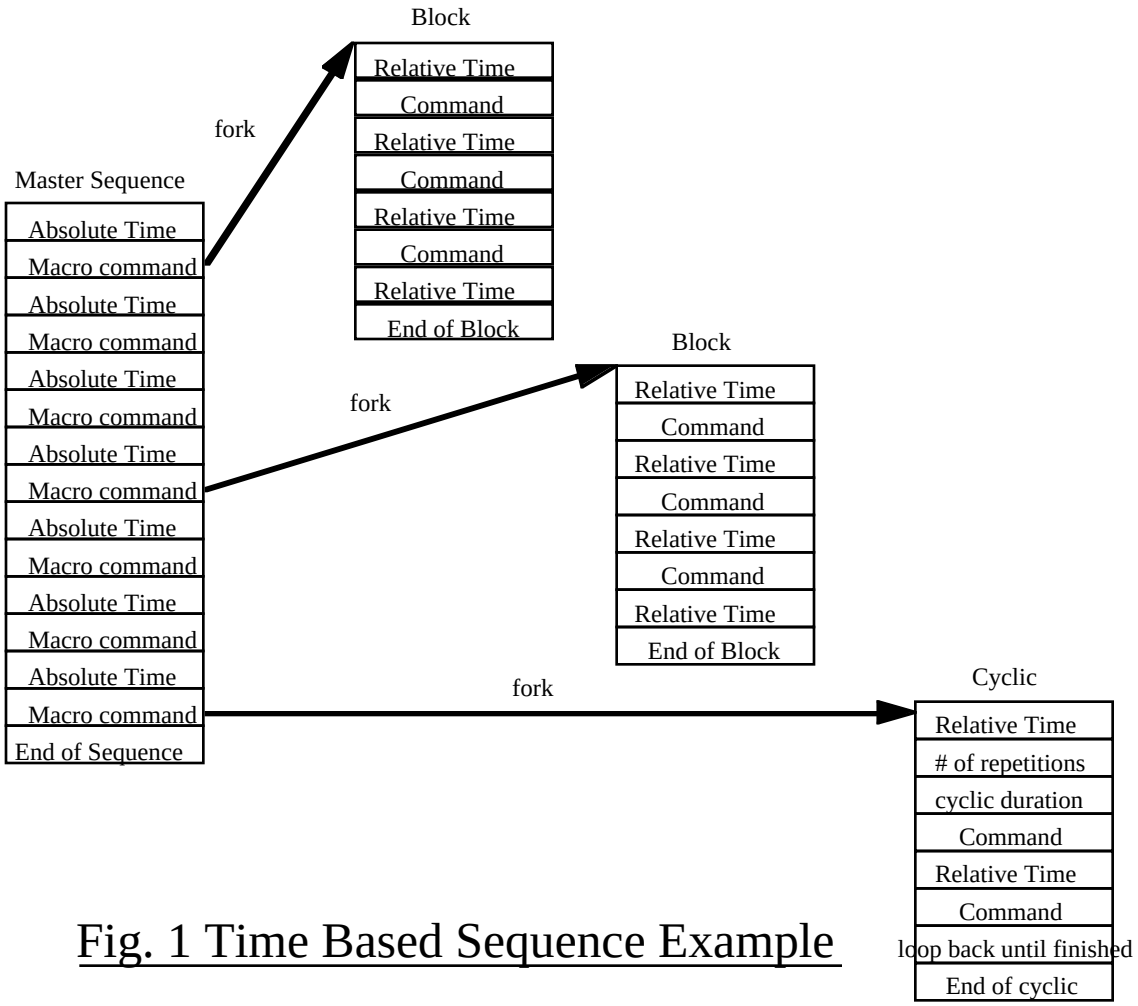


Fig. 1 Time Based Sequence Example

Sequence Components

Sequences:

This construct is the traditional 'sequence' (see fig. 1) used on Voyager, Galileo, MGS, etc. A series of commands / macro commands (to initiate blocks) is uplinked to the spacecraft at a planned window to execute over the next few days, weeks, or months. Very long sequences are called 'background sequences', and are designed to work in concert with other shorter 'overlay sequences' (e.g., mini-sequences).

Blocks:

Candidate activities for blocks are repetitive and might include science maintenance activities, engineering calibrations, and/or engineering maintenance activities. Blocks can be called from the stored sequence macro commands. Parameters may be passed with the

call to the block for use during execution. The MGSO 'brick' concept and Spacecraft Expanded Blocks (SEB) belong to this class of sequence construct.

Mini-sequences:

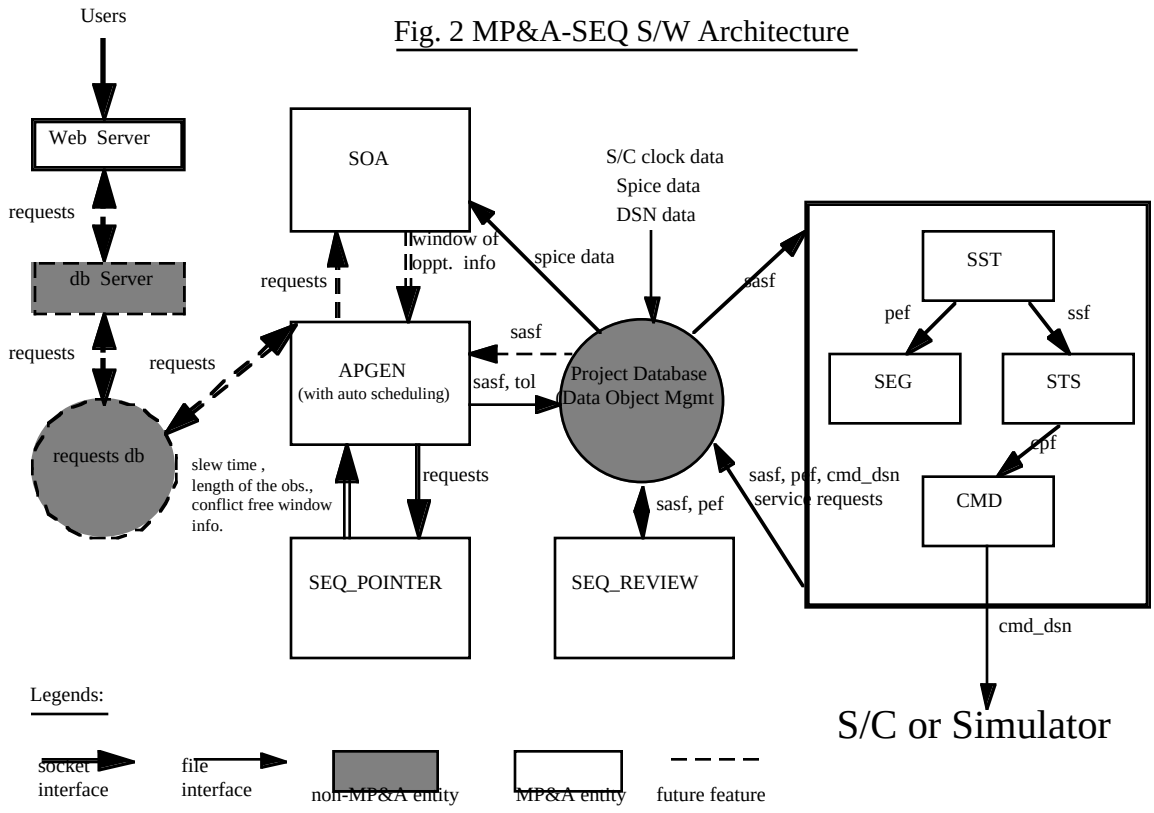
Mini-sequences are similar to ordinary sequences except that they are generally shorter. TCMs will be implemented using the mini-sequence strategy. For example, a window will be designed in a stored sequence that encompasses an execution period for a TCM. The sequence will be uploaded without the TCM and begin execution. The TCM (which may require a late parameter update) will then be designed on a separate schedule and uplinked to execute in parallel with the stored sequence.

Cyclics:

Cyclics are used to sequence repetitive activities that are defined as a repeatable command set once in a sequence and then executed a number of times (Voyager cyclics are the best historical example of the construct being proposed in this document). As a result, a cyclic can only be called while the sequence containing the cyclic definition is executing. Cyclic parameters are the number of repetitions and the amount of time between repetitions.

Planning & Sequence Ground S/W (fig. 2 MP&A-SEQ S/W Architecture)

Fig. 2 MP&A-SEQ S/W Architecture



Science Planning System

SEQ_POINTER:

Allow user to design remote sensing science observations during a flight operations: i) to reconnoiter candidate remote sensing observations, ii) to design and refine remote sensing observations, and iii) to update observation targeting based upon improved ephemeris knowledge.

Science Opportunity Analysis Tool:

SOA will identify and select optimal science opportunities to support both trajectory selection and sequence development. SOA will identify time windows during which a given set of geometric conditions that define a science opportunity are met. SOA will analyze a fixed time window and gives the values of geometric parameters versus time. Finally, SOA will animate (computer graphic visual) moving representations of the target.

Mission Planning & Sequencing System

APGEN:

APGEN is a resources -based interactive activity plan generator for mission planning & sequencing, which allows automatic scheduling and modeling of activities / resources.

SST:

SST provides a capability to create, merge, edit, print, expand and check sequence requests. First, SST will validate sequence request parameter values with respect to its type and ranges, as well as to the constraints governing relations between different parameters. Second, the program will also expand the sequence requests into one or more lower levels sequence activities, notes or commands. Finally, SST will verify sequences are consistent with the flight and mission rules, finite S/C and ground resources,. To accomplish these roles, SST will update and maintain models of both S/C and ground states in order to check whether allocated constraints are violated by a candidate sequence.

STS:

SST is responsible for the translation of a Spacecraft Sequence in the form of a Spacecraft Sequence File (SSF) into a Command Packet File (CPF) for radiation to the spacecraft. A binary UNIX file may be formatted into a CPF for transmission to the spacecraft. The primary program of the STS is the Spacecraft Language Interpreter and Collector (SLINC) which is based on the prototype program Seqtran_2000..

SEQ_Review:

SEQ_Review is similar to a text editor that lets the user open an arbitrary text file for display. Unlike a text editor, SEQ_Review can be told to detect certain types of file formats. When it finds that a file conforms to a format it "knows", SEQ_Review analyzes the file in considerable detail. This allows the user to modify the appearance of the file (e. g., remove unwanted information, re-format data into columns, add derived quantities computed from data in the file) much more easily than with a text editor. SEQ_Review learns about file formats through ASCII files called "Format Descriptors" (FD). These files can be edited and modified by users and adapters, providing SEQ_Review with great flexibility.

11.6.4 Phase 2: Close Loop Conditional Sequence (see fig. 3 time based conditional sequence example)

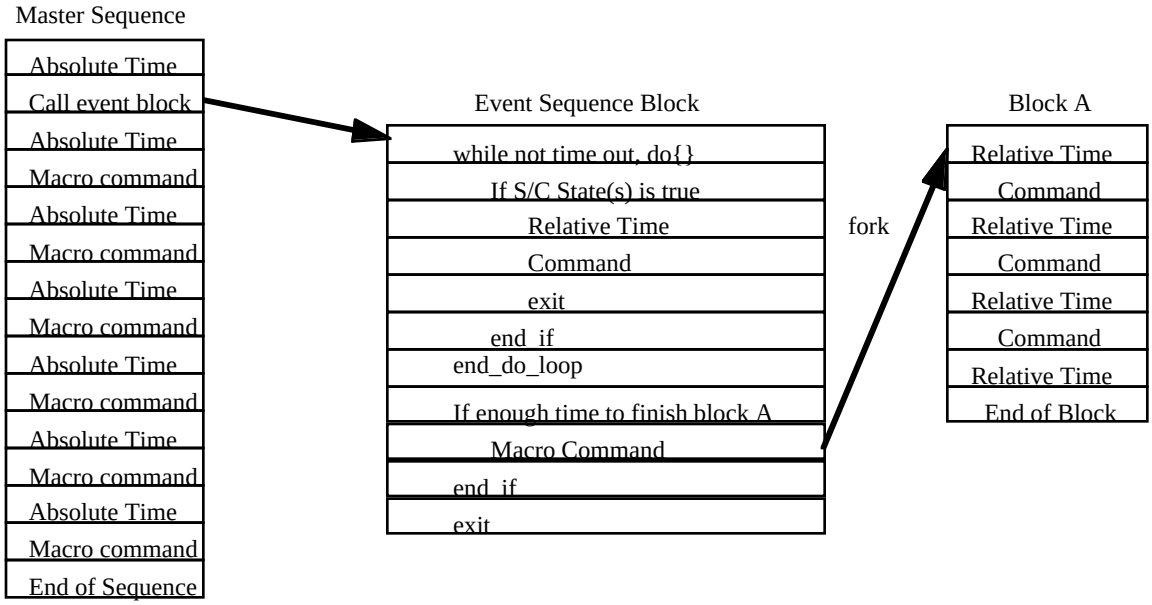


Fig. 3 Close Loop Conditional Sequence Example

This addition layer allows the traditional Time Based Sequence to perform closed loop Sequence. A series of commands / macro commands (to initiate sequence blocks or event blocks) is uplinked to the spacecraft at a planned window to execute over the next few days, weeks, or months. An event block will be initiate at an absolute time, but the commands will be executed depends upon the conditions of a selected set S/C state(s). The conditions of the states are the actual S/C states during sequence execution time, and it does not require any ground interactions.

Planning & Sequence Ground S/W

In addition to the MP&A-SEQ S/W (see fig. 2) from phase 1, an external file is needed to set the S/C states conditions.

11.6.5 **Phase 3: Event Driven Sequence** (see fig. 4 event driven sequence example)

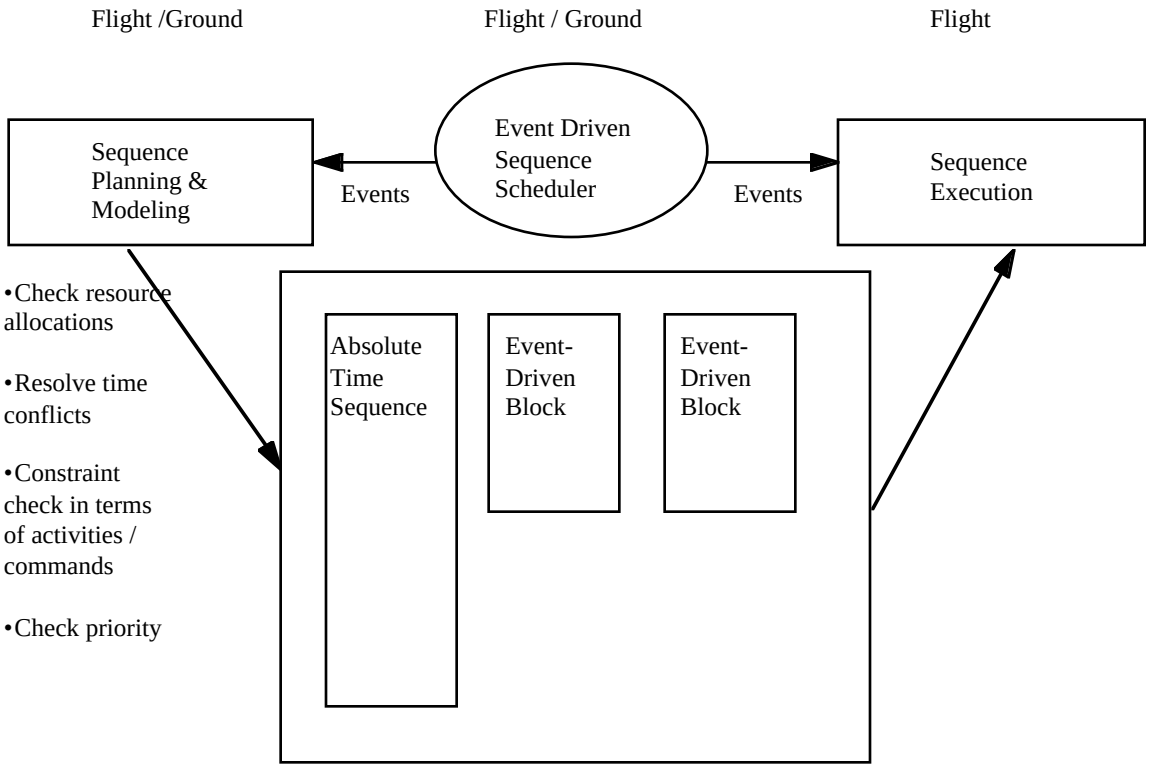


Fig. 4 Event Driven Sequence

The unified flight ground S/W architecture plays an important role in this particular phase, Sequence planning & modeling S/W and Event driven sequence scheduler will be the identical S/W used both on the ground and on-board the S/C.

Both the absolute time sequence and event driven sequences / blocks are developed on the ground by the sequence planning & modeling, and event driven scheduler tools. The sequence planning & modeling S/W will schedule the absolute time sequence w.r.t. the sequence activities' priority, and constraint check both the absolute time and event driven sequence. The event driven scheduler will model the S/C states for the event driven sequences / blocks. After the sequences' uplink, the absolute sequence will be executed as time expired, whereas the event driven blocks are triggered by the elected S/C states via the on-board event driven scheduler. Prior to the execution of the event driven blocks, the on-board sequence planning & modeling S/W will check the S/C resource allocations, check priority, resolve any time conflicts with the overall sequence, and perform sequence constraint checking.

11.7 Development Environment

1.0 Introduction

This section of the software architecture document addresses the configuration, facilities, and tools necessary for development of the X2000 Unified Flight Ground software systems. The highest level guiding paradigms are:

1. Create a repeatable process
2. Approach the design & development process with a “tool based” approach
3. Increase programmer productivity

2.4 Design

The creation of the X2000 design will be with the Unified Modeling Technique. The choice of what model projections' one creates has a profound influence upon how a problem is approached and how a solution is shaped. Abstraction, the focus on relevant details while ignoring others, is a key to learning and communicating. Because of this:

- Every complex system is best approached through a small set of nearly independent views of a model; no single view is sufficient.
- Every model can be expressed at different levels of fidelity.
- The best models are connected to reality.

In terms of the views of a model, the UML defines the following graphical diagrams:

- Use case diagram
- Class diagram
- Behavior diagrams
- State diagram
- Activity diagram
- Sequence diagram
- Collaboration diagram
- Implementation diagrams
- Component diagram
- Deployment diagram

These diagrams provide multiple perspectives of the system under analysis or development. The underlying model integrates these perspectives so that a self-consistent system can be analyzed and built. These diagrams, along with supporting documentation, are the primary artifacts that a modeler sees, although the UML and supporting tools will provide for a number of derivative views.

2.0.1 Notation and Semantics History

The UML is an evolution from Booch, OMT, OOSE, most other object-oriented methods, and many other sources. These various sources incorporated many different elements from many authors, including non-OO influences. The UML notation is a melding of graphical syntax from various sources, with a number of symbols removed (because they were confusing, superfluous, or little-used) and with a few new symbols added. The ideas in the UML come from the community of ideas developed by many different people in the object-oriented field. The UML developers did not invent most of these ideas; their role was to select and integrate ideas from the best OO and computer-science practices. The genealogy of the notation and underlying detailed semantics is complicated, so it is discussed here only to provide context, not to represent precise history.

Use-case diagrams are similar in appearance to those in OOSE.

Class diagrams are a melding of OMT, Booch, class diagrams of most other OO methods. Process-specific extensions (e.g., stereotypes and their corresponding icons) can be defined for various diagrams to support other modeling styles.

Statechart diagrams are substantially based on the statecharts of David Harel with minor modifications. The Activity diagram, which shares much of the same underlying semantics, is similar to the work flow diagrams developed by many sources including many pre-OO sources.

Sequence diagrams were found in a variety of OO methods under a variety of names (interaction, message trace, and event trace) and date to pre-OO days. Collaboration diagrams were adapted from Booch (object diagram), Fusion (object interaction graph), and a number of other sources.

Collaborations are now first-class modeling entities, and often form the basis of patterns.

The implementation diagrams (component and deployment diagrams) are derived from Booch's module and process diagrams, but they are now component-centered, rather than module-centered and are far better interconnected.

Stereotypes are one of the extension mechanisms and extend the semantics of the meta-model. User-defined icons can be associated with given stereotypes for tailoring the UML to specific processes.

The scope of the UML tools will cover the design needs of X2000 software designers. There are several Computer Aided Software Engineering (CASE) tools and companies that provide consultations services for UMT that would support a quick start for the creation of the X2000 software design.

3.0 X2000 Design, Development, and Operational Resources

The design and development platform will be PC based. Initially, the Windows NT operating systems will be used by all software designers and developers. It will also be the target operational platform for the X2000 ground data system. The performance/cost index between PC’s and UNIX workstations is significant. PC performance is roughly equivalent to UNIX workstations. However, the purchase price of PC’s is significantly less than UNIX workstations and should constitute a significant saving in both hardware and system software procurements for X2000. The Windows NT operating system installation base now exceeds UNIX. Windows NT has matured and constitutes a solid

development and operational platform. Current NT architecture features and future plans constitute significant flexibility for X2000 hardware configurations.

The embedded flight system development will be hosted on PC's and will cross-compile to the Power PC Architecture(or whatever target is decided upon by the X2000 Hardware Architecture Committee). Flight system developers will use some incarnation of the VX Works real time operating system (RTOS) developed by Wind River Systems. This RTOS supports the requirements for single and distributed flight software configurations.

3.0.1 Languages

The X2000 language philosophy will be the "right tool for the job." However, some high-level guidelines will be applied to language selection for ground data system development:

- It should be capable of persistence operations with an OODBMS
- It should have bindings to an interface or object definition language (IDL or ODL)

Flight system development, language selection should be predicated upon:

- Availability of supporting "run-time" system
- Ability to process real-time deadlines
- Ability to interface with languages capable of supporting real-time events if it is not
- capable on it's own (e.g. C++/Assembly Language)
- Supported language of VxWorks RTOS
- Preferably, Object Oriented in Nature.

3.0.3 Databases

X2000 will make exclusive use of Object Database Technology. The whole system design, development,

and test strategies are Object Oriented in nature. Therefore it logically follows that the persistence mechanism should be predicated upon the aforementioned. Using tables to store objects is like driving your car home then disassembling it to put it into the garage. It can be assembled again in the morning, but one eventually asks whether this is the most effective way to park a car.

3.0.4 Frameworks

To achieve rapid development and aggressive delivery schedules for X2000 systems, the prevailing paradigm in this aspect of system development is, the cheapest line of code is the line of code you don't have to write. Frameworks within X2000 will be classified into two areas:

- Desktop
- Middleware

The elements of a desktop framework are as illustrated:

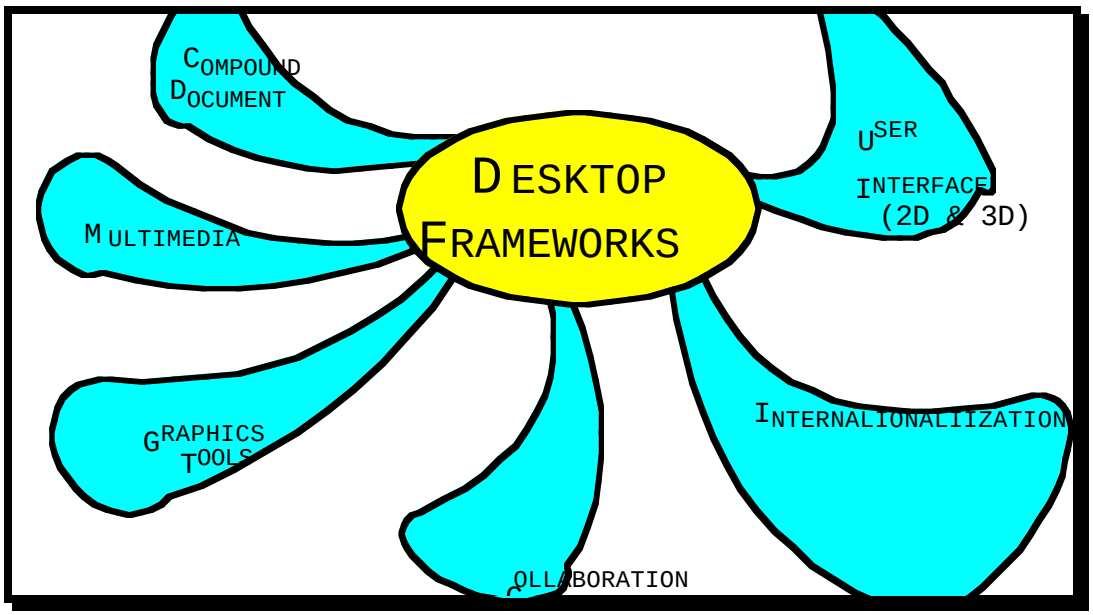


Figure 5 - Desktop Framework Mind-Mapping Diagram

Desktop frameworks should both increase X2000 programmer productivity and change the way X2000 visual applications are developed. The result is a superior end-user application interface at a lower cost.

The elements of a middleware framework are as illustrated:

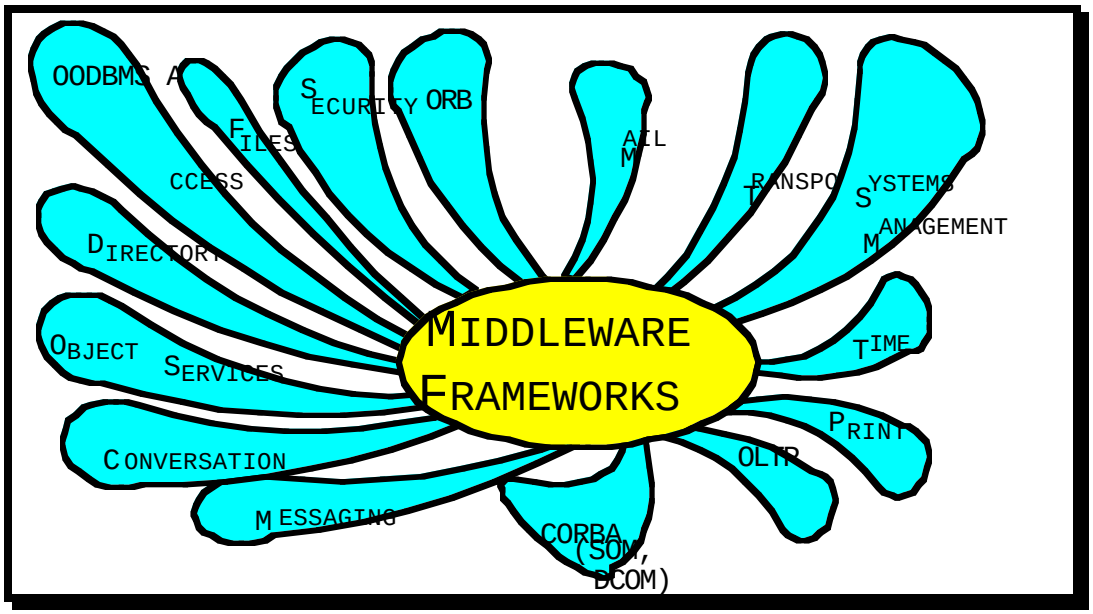


Figure 6 - Middleware Framework Mind-Mapping Diagram

Frameworks will improve the ability of X2000 developers to deal with the complexity of distributed development. They will help coordinate middleware elements that run on

distributed heterogeneous platforms. Frameworks will create an infrastructure of communicating system objects. Where each system object is defined by using an Object Request Broker's (ORB) IDL. Each system will be self describing. The best part is, the components are certified to work together. X2000 developers simply provide the code that customizes the system at a very fine grained level.

4.0 Integrated Development Environments

X2000 developers will achieve significantly more productivity by using Integrated Development environments. The environment for X2000 support must have:

1. Incremental Compilation and Linkage
2. Source Level Debugging
3. Help
4. Developer Support Tools
5. Make Capability
6. Syntax coloring
7. Support different editors (VI, EMACS, Brief, etc..)
8. Support 3rd party tool "plug-ins"
9. Support Code Browsing
10. Integrated configuration management
11. Integrated WWW access
12. Resource Editors (for creation of icons, bitmaps, etc.)

The key idea is to have a single tool for developer's to learn. Third party tools must be able to operate in this environment. Productivity will be maximized keeping developers within their tool focused on quality code production.

11.8 Java and Java Beans: A Component Architecture for Java at JPL

Abstract

Over the past few years, constructing applications by assembling re-usable software components has emerged as highly productive and widely accepted way to develop custom applications. First generation products such as Microsoft's Visual Basic, with its VBX components and "forms-based" application assembly process proved to be very useful in building a broad spectrum of applications. Visual Basic has since been followed by other products such as Borland's Delphi which further enhanced the basic component assembly application development model by adding more powerful data access components

and object-oriented component extension capabilities. Now, JPL developers building an institutional software infrastructure for the Outer Planets project can leverage the most modern and adaptable software development tool ever created for a component based flight and ground system software architecture. It can not be stressed enough that the components that are being developed, extended, and reused are in a binary incarnation.

Java and Java Beans take the **component software** assembly paradigm to a new level. Java Beans is an architecture and platform neutral API for creating and using dynamic Java components. Java Beans build on the strengths of the **component assembly development** model established by these pioneering products, and extends the power further. Application developers will be able to use a variety of development tools to assemble custom applications from fully portable Java Beans. This document is a brief overview of Java and Java Beans and its functional capabilities. It discusses:

- How Java Beans extend and enhance the capabilities of the portable Java Platform
- The key elements that make up a software component model
- Highlights of Java Bean functional capabilities
- How Java Beans Extend the Java Platform

Building on Java Strengths

Java has quickly established itself as the industry standard platform for building fully portable Internet and Corporate Intranet applets and applications. The Java platform provides a number of advantages to developers for these types of applications:

- **Fully portable platform**: language, libraries and virtual machine pervasive presence of the Java platform in Browsers, and **soon within Operating Systems** (soon means in terms of embedded OS, Wind River should have a Java port FY96Q4 for VXWorks) themselves, allows developers to write application functionality once and deploy the application broadly on a wide variety of OS and hardware platforms.
- **Powerful and compact environment**: The Java platform provides developers with the full power of an object-oriented language while eliminating the complexity, housekeeping and heavier weight object creation and registration processes required

by other language and programming model environments. **The lightweight runtime can be incorporated in chips for embedded systems, in PDAs as well as client and server class PC's and workstations where Java is becoming increasingly pervasive**.

- **Network aware**: From its inception, the Java platform has been network aware. TCP/IP support is built in. **Security mechanisms** which allow full protection from applet interference with client-side data are built-in. Finally, the platform was designed to allow applets and applications to be built from self-describing classes which can be easily downloaded to the client-environment without the need for heavy weight installation or registration processes. Java Beans build on all of these strengths and extends the Java platform further.

Component Model Overview

Before describing the services provided by the Java Beans API, it is useful to have a high level understanding of the key elements and services provided by component models in general.

Component Model Elements

A **component model** is an architecture and set of APIs that allow developers to define software components that can be dynamically combined together to create an application. A component model consists of two major elements: **components and containers**.

Components can range in size and capability from small GUI widgets like a button, to applet size functionality such as a tabular viewer to a more full sized application such as an HTML browser of a text layout application. **Components** can have a visual appearance such as a button, or can be non-visual, such as a telemetry data feed monitoring component.

Containers are used to hold an assembly or related components. Containers provide the context for components to be arranged and interact with one another. Containers are sometimes referred to as forms, pages, frames or shells. Containers can also be components, i.e. a container can be used as a component inside another container.

Component Model Services

A component typically provides five major types of services:

- Component Interface Publishing and Discovery
- Event Handling
- Persistence
- Layout
- Application Builder Support

These are described in greater detail below.

Runtime Component Interface Publishing and Discovery

This is the mechanism for components to "publish" or "register" their interfaces so that they can be "driven" dynamically by calls and event notifications from other components or application s. In ground station example, this publishing and discovery mechanism is what allows the telemetry point component to ask the strip charting component to draw a graph of its data. Since the strip charting component has "registered" its interfaces to the component environment, the strip charting component does not need to be of the same application "build" as the telemetry point component. Instead, even though they were built

separately, they can interact in a dynamic way using the services provided by the component environment.

Event Handling

Event Handling is the mechanism for components to "raise" or "broadcast" events and have those events delivered to the appropriate component (s) that need or want to be notified. Notified components in turn typically perform some function. For example, if the developer of the control center application provided a button for the user to select between a CVT chart or a line graph, the event handling systems would notify the charting component when the user clicked on the button component.

In addition to system events such as clicking the mouse, components can define their own events. For example, a component that monitors a live data feed such as telemetry information or the power status of a antenna or a change in the data rate of the FEP. The "data changed" event could be handled by a variety of other components to sound an alarm, change a visual display or start another process.

Persistence

Persistence is the mechanism for storing the state of components in a non-volatile place. Component state is stored in the context of the container and in relationship to other components. For example, if the Principle Investigator wanted to save the on-line science web page with their telemetry health information and chosen charts, the persistence mechanism would support this.

Layout

There are two major types of layout control which component models support. First, they provide a way for a particular component to control its visual appearance within its own space. Second, component models provide mechanisms and services for a component's layout in relation to other components inside a container. This includes services for handling appropriate behavior when the component is activated. For some types of components this may include such things as menu bar merging.

For the great majority of applications, layout requirements for components in the context of a container are straight forward. Most component layout requirements are satisfied by giving each component a non-overlapping rectangular space. Developers control the lay out the components (buttons, viewers etc.) in a logical, easy to use manner that supports the application's functionality. End users do not rearrange the components at runtime.

Application Builder Support

Application Builder Support interfaces enable components to expose their properties and behaviors to Application Builder Development tools. Using these interfaces, tools can determine the properties and behaviors (events and methods) of arbitrary components. The tools can the provide mechanisms such as tool palettes, inspectors, editors which the application developer uses to work with the various components to assemble an application. Through these mechanisms the application developer can modify the state and appearance of components application developer can modify the state and appearance of components as well as establish relationship between components.

For example, let's look at the button and charting components from our on-line banking example. Recall that when the end-user presses the button the chart switches from a bar chart to a line graph. When assembling this application, the application developer uses property editors to specify the appearance (size, color, label) of the button and the default type of chart (bar chart) to display. The developer uses other application development tool mechanisms to specify the relationship between the button's "click" event and the chart component's "chart type" property.

Distributed Computing

In addition to these five major services, component models often provide a strategy for using the components in a **distributed computing environment**. For example, a component that monitors the status of an encounter sequence may run on a server attached to that machine. If a data value changes, the server component may raise an event which gets delivered over the network to another software component running on a PI's desktop machine. The desktop component could then respond appropriately, perhaps posting a message, or changing the shape of a graph.

Clearly, there is a big difference between software components interacting on a single computer and components interacting over a network. Besides needing to take into consideration the slower speeds of a network, the developer and distributed computing infrastructure also need to provide appropriate recovery and re-synchronization mechanisms should either component or machine fail in some fashion.

Attempting to simply extend a single machine desktop component model to encompass all the requirements demanded by complex heterogeneous networked computing environments is fraught with problems and trade-offs. The desktop component model will either become burdened with more complex APIs and heavier weight execution environments, or the distributed computing capabilities will be less than robust.

Consequently, component models that are serious about providing full distributed computing capabilities will leverage robust established distributed computing technologies such as CORBA. This way the single machine component model can be kept compact and light weight, while also providing access to rich functionality that may be required by distributed applications.

The next section of this document discusses Java Beans in the context of the elements and services that comprise a software component model.

Java Beans API Highlights

Java Beans is an architecture and platform neutral API for building and using dynamic Java component and container functions, and provides assess to the five major component model services outlined above, namely:

- Component Interface Publishing and Discovery
- Event Handling
- Persistence
- Layout
- Application Builder Support

Java Beans component model services can be implemented by bridging to specific component models, including Microsoft's OLE/COM, CI Lab's OpenDoc, Netscape's Live Connect. In addition Java Beans will run on **JavaSoft's embeddable JavaOS. Libraries bridge the Java Bean API to the various component model implementations**.

Thus, a developer can build components completely in Java using fully portable Java Beans APIs. Developers will not have to intersperse non-portable platform or component model specific calls in their portable Java code. The Java platform (which will include the Java Beans APIs) allows component developers to write both the functional capabilities and component behavior aspects of a component completely in Java.

Java Beans components raise the Java notion of **"write once, use anywhere"** to a new level. Java Beans integrate in a high quality way into a variety of containers, including Netscape (using Javascript and LiveConnect), in HotJava and other Java containers, and Microsoft containers (such as Explorer, Visual Basic, Windows Shell, and Word), OpenDoc containers, and OLE containers such as PowerBuilder, Delphi and other visual builder tools that support OLE/COM.

Java Beans services will be part of the Java platform. This means that developers **will not need to distribute any extra libraries in order for applets and applications built using Java Beans to work**. In addition, Java Beans will be able to be used outside of containers as independent Java applets which can communicate dynamically.

The **Java Beans platform and architecture neutral API in combination with the fact that a Java component's full functional implementation is also fully portable make Java Beans highly re-useable**. Unlike with other component models, Java Beans are not bound to a particular platform or container or component model. Consequently, JPL developers of Java Beans will be able to target all future missions.

Consumers of components (e.g JPL Flight and Ground Systems Developers) will get the highest leverage from the purchase and time spent learning to use a Java Bean component. They will be able to re-use Java Beans in a wide variety of Internet, Intranet and even proprietary client/server applications.

The following section provides some insight into the design goals and characteristics developers can expect to see in the Java Beans API.

Design Goals

The major design goals of the Java Beans API and implementation approach are:

Java Beans is compact and easy to create and use. Java Beans can be very compact and simple to create. In particular the simple components will be very easy. Building more complex components in Java will be possible as well. By fully leveraging the Java platform's functionality they can also be kept very compact.

Existing component model APIs have emerged by scaling down from complex heavy weight application size component to include lighter weight widget size components. In contrast, Java Beans APIs have been designed by scaling up from simpler lighter weight widgets, applets, and applications towards more full function applications. Consequently, the Java Beans API will not overburden the smaller widget and applet size components with complexity and weight. Since these smaller sized components are most prevalently used, Java Beans' compact design will be advantageous to both component builders and component consumers.

Java Beans is fully portable.

Java Beans are fully portable through the platform neutral Java Beans API and bridging libraries that will be part of the standard Java platform. As a result, developers will not need to include non-portable code or be concerned with including platform-specific libraries with their Java applets.

Java Beans leverage many of the inherent strengths of the Java Platform. Java Beans takes advantage of the existing class discovery mechanism already built into the Java platform. This mechanism uses Java's unique introspection and reflection technology. This means that Java does not need to introduce additional, heavier

weight registration mechanisms to the runtime to support interface publishing and discovery.

In addition, consistent with Java's overall design center of using light weight and easily understandable mechanisms, many Java Beans will not require any additional programming by the developer. For example, Abstract Window Technology (AWT) components will be Java Beans automatically.

The Java Beans libraries will also provide default component behaviors for simple components. For example, automatic persistence will be handled using Java serialization. In addition Java Beans will provide automatic generation of property editors by examining a component's get and set methods. Of course, component developers will be able to override the default behavior as may be required by more complex components or a desire to provide richer component editors. Finally, Java Beans components will also benefit from the new AWT desktop integration capabilities such as cut-copy-paste and drag-and-drop interfaces.

Java Beans leverages robust distributed computing mechanisms. Java Beans component model APIs and implementation are focused on components interacting in a single virtual machine. Rather than over complicating the Java Beans API or burdening the Java platform with heavier weight distributed computing

mechanisms, Java developers will be able to chose among several distributed computing approaches. For example, developers will be able to add distributed component interaction

to their Java applications by using Java's Remote Method Invocation, by using industry standard CORBA IDL interfaces for remote object access, or by other distributed computing mechanisms. Developers will be able to chose the mechanism that best suits their portability, performance and legacy integration requirements.

Flexible build-time component editors.

Java Beans will allow component developers to specify a broader variety of build-time property sheets, inspectors and editors for their components. This will allow developers to provide the most productive way for component users to reap the full value form component capabilities. For example, a data base connection component provider might want to provide more than a long, complicated property sheet for the developer to use at build time. Instead the component provider might want to organize the various properties in tabbed sections or perhaps allow the component user to visually specify table joins at build time. The Java Beans Application Builder Tool APIs will support a way for component developers to create the best type of property editor for their component type.

Conclusion

Java Beans further enhance the portable Java platform by adding new levels of dynamism, flexibility and re-use. Java Beans take the **component assembly model of application development to a new level**. Java Beans are compact, easy to build, fully portable, and re-useable in the broadest number of containers and environments. Java and Java Beans will enable JPL developers powerful and exciting new types of embedded applications for space missions, “mission oriented” Intranet applications and Internet inter-activity for the science and academic communities.

11.9 JTAG testability

Flight Hardware Test/BIST

JTAG IEEE 1149.1 Standard

With increased silicon complexity and shrinking IC packaging geometry, board interconnect validation has become difficult and expensive using conventional PCB and MCM testing. The IEEE 1149.1 standard's primary objective is to enable system users to control and observe a device's input and output pins for the purpose of interconnect testing. This protocol also ensures interoperability between components in a system.

The implementation of a JTAG testability architecture in a flight system would not only allow circuit and interface verification at the IC, MCM, and PCB levels, but also the Systems level, while always allowing for future expansion. It could also be used for high resolution onboard flight heath diagnosis/determination, or even function as an additional Systems level data bus should it ever became necessary.